Міністерство освіти і науки України
Харківський національний університет радіоелектроніки
Факультет комп'ютерних наук
Кафедра інженерії програмного забезпечення

Звіт
з лабораторної роботи №3
з дисципліни «Теорія паралельних обчислень»
на тему «Паралельні алгоритми пошуку підпослідовностей у текстових даних»

Виконали ст. гр. ІПЗм-22-6:
Миронюк С.А.,
Сєнічкін І.О.

Перевірив викладач:
доц. Кобзєв В.Г.

Харків, 2023

ПАРАЛЕЛЬНІ АЛГОРИТМИ ПОШУКУ ПІДПОСЛІДОВНОСТЕЙ У ТЕКСТОВИХ ДАНИХ

Мета – навчитися створювати і аналізувати ітераційні паралельні алгоритми виявлення підпослідовностей в однорідних текстових безперервних потоках даних для різних програмних середовищ.

Індивідуальне завдання:

На вході маємо великий набір генетичного матеріалу, для якого необхідно з'ясувати наявність в ньому підпослідовностей, що є маркерами хвороби А.

1. Розробити послідовний та паралельний алгоритми пошуку відповідних маркерів:
   а) наївним методом;
   б) методом Ахо-Корасіка;
   в) методом Бойера-Мура;
   г) методом Рабіна-Карпа;
   д) методом Shift-Or.
2. Здійснити аугментацію запропонованих цільових маркерів.
3. Обчислити значення показників прискорення та ефективності розпаралелювання при збільшенні розміру кількості маркерів від 100 до 10000 (при зміні на порядок).
4. Порівняти отримані результати для декількох різних мов програмування (або різних підходів до паралелізації).
5. Зробити висновки щодо доцільності запропонованого паралелізованого підходу.

Варіант: в) метод Бойера-Мура

Вхідні дані: дані псевдогеномів pseudo10006.fasta. (https://1001genomes.org/data/GMI-MPI/releases/v3.1/pseudogenomes/fasta/). Для здійснення аугментації запропонованих цільових маркерів було узято файл sample_markers.csv.

Хід роботи:

Алгоритм Бойєра-Мура, розроблений двома вченими – Бойєром (Robert S. Boyer) та Муром (J. Strother Moore), вважається найшвидшим серед алгоритмів загального призначення, призначених для пошуку підрядку у рядку. Рядок – вся послідовність символів тексту. Це не обов'язково про текст. В загальному випадку рядок – це будь-яка послідовність байтів. Пошук підрядку у рядку здійснюється за заданим зразком, тобто деякою послідовністю байтів, довжина якої не перевищує довжину рядка. Наше завдання полягає в тому, щоб визначити, чи містить рядок заданий зразок.

Найпростіший варіант алгоритму Бойєра-Мура складається з наступних кроків. На першому кроці будуємо таблицю зсувів для зразка. Далі поєднуємо початок рядка та зразка та починаємо перевірку з останнього символу зразка. Якщо останній символ зразка та відповідний йому при накладенні символ рядка не збігаються, зразок зсувається щодо рядка на величину, отриману з таблиці зсувів, і знову проводиться порівняння, починаючи з останнього символу зразка. Якщо ж символи збігаються, проводиться порівняння

передостаннього символу зразка і т. д. Якщо всі символи зразка збіглися з накладеними символами рядка, то ми знайшли підрядок і пошук закінчено. Якщо якийсь (не останній) символ зразка не збігається з відповідним символом рядка, ми зрушуємо зразок на один символ праворуч і знову починаємо перевірку з останнього символу. Весь алгоритм виконується до тих пір, поки не буде знайдено входження шуканого зразка, або не буде досягнуто кінця рядка.

Величина зсуву в разі розбіжності останнього символу обчислюється виходячи з таких міркувань: зсув зразка повинен бути мінімальним, таким, щоб не пропустити входження зразка в рядку. Якщо цей символ рядка зустрічається у зразку, ми зміщуємо зразок таким чином, щоб символ рядка співпав із правим входженням цього символу у зразку. Якщо ж зразок взагалі не містить цього символу, ми зрушуємо зразок на величину, рівну його довжині, так що перший символ зразка накладається на наступний символ рядка, що перевірявся.

Величина усунення для кожного символу зразка залежить тільки від порядку символів у зразку, тому зміщення зручно обчислити заздалегідь і зберігати у вигляді одновимірного масиву, де кожному символу алфавіту відповідає зміщення щодо останнього символу зразка. Пояснення на прикладі: Нехай у нас є набір символів із п'яти символів: a, b, c, d, e і ми хочемо знайти входження зразка “abbad” у рядку “abeccacbadbabbad”. Наступні схеми ілюструють усі етапи виконання алгоритму:

a	b	c	d	e
1	2	5	0	5

Таблиця зсувів для зразка “abbad”.

```
a b e c c a c b a d b a b b a d
a b b a d
```

Початок пошуку. Останній символ зразка не збігається з накладеним символом рядка. Зсуваємо зразок вправо на 5 позицій:

```
a b e c c a c b a d b a b b a d
          a b b a d
```

Три символи зразка збіглися, а четвертий – ні. Зсуваємо зразок вправо на одну позицію:

```
a b e c c a c b a d b a b b a d
            a b b a d
```

Останній символ знову не співпадає із символом рядка. Відповідно до таблиці зсувів зрушуємо зразок на 2 позиції:

```
a b e c c a c b a d b a b b a d
                a b b a d
```

Ще раз зрушуємо зразок на 2 позиції:

```
a b e c c a c b a d b a b b a d
                  a b b a d
```

Тепер, відповідно до таблиці, зрушуємо зразок на одну позицію, і отримуємо входження зразка:

a b e c c a c b a d b a b b a d
a b b a d

Цей алгоритм непогано підходить для обробки довгих текстів, але не виправдовує себе на коротких. За наявності рядка S довжиною m і підрядку довжиною n метод показує складність O(n/m) при сприятливих, проте дає непогані результати навіть на «невдалих» текстах маючи складність O((n-m+1)*m+p) де p – потужність алфавіту.

Дана програма була виконана на пристрої DESKTOP-IDO8GRU з процесором Intel(R) Core(TM) i5-7600 CPU, який включає 4 ядра, базова тактова частота 3500 МГц, об'єм кеш пам'яті 3 рівня (L1 – 256 КБ, L2 – 1,0 МБ, L3 – 6 МБ).

Програма була реалізована на мовах програмування Java і Python.

Імпортуються необхідні бібліотеки:

java.io.BufferedReader: Використовується для читання текстових даних з вхідного потоку з використанням буфера для забезпечення ефективного читання.

java.io.FileReader: Надає засоби читання з файлу.

java.util.ArrayList: Реалізація динамічного масиву для зберігання маркерів та результатів пошуку.

java.util.List: Інтерфейс, який використовується для представлення списку маркерів та результатів.

java.util.concurrent.ForkJoinPool: ця бібліотека містить клас ForkJoinPool, який є частиною Java Fork/Join Framework. Він використовується для паралельного виконання завдань у вигляді "розгалуження та об'єднання" (fork/join) з метою покращення продуктивності на багатоядерних процесорах.

java.util.concurrent.RecursiveTask: Абстрактний клас, що надає базову реалізацію для створення паралельних завдань із поверненням результату.

import concurrent.futures (Python): Модуль для роботи з паралельними обчисленнями

import time (Python): Модуль для вимірювання часу

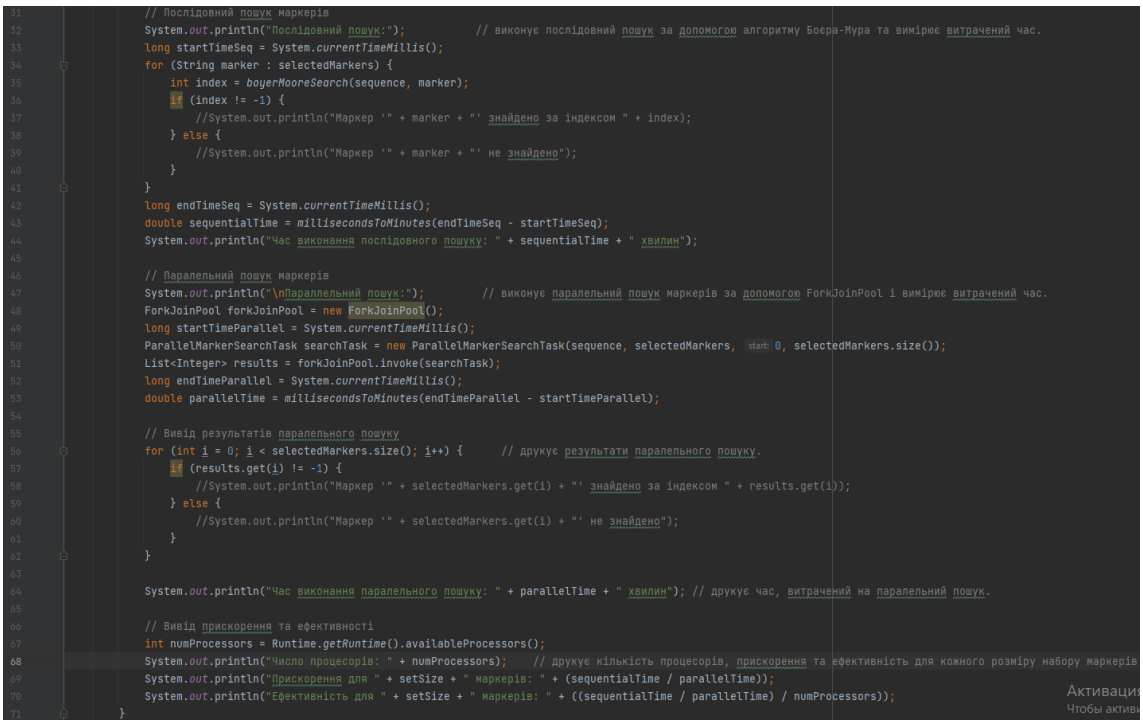
Програмна реалізація алгоритму Бойєра-Мура на мові програмування Java.

В першій частині коду виконується читання даних з файлів: CSV-файл, FASTA-файл.

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class FileProcessor {                    // оголошення класу: клас FileProcessorогоголошено

    public static void main(String[] args) {        // Основний метод: точка входу в програму
        // Шляхи до файлів
        String csvFilePath = "G:/Сергей/XHYPE/3 семестр/Параллельные вычисления/Lab 03/sample_markers.csv";     // Шляхи до файлів: рядки, що представляють шляхи до файлів CSV і FASTA
        String fastaFilePath = "G:/Сергей/XHYPE/3 семестр/Параллельные вычисления/Lab 03/pseudo10006.fasta";

        // Читання маркерів із CSV-файлу
        List<String> markers = readMarkersFromCSV(csvFilePath); // викликає readMarkersFromCSVметод читання маркерів із файлу CSV.

        // Читання послідовності з FASTA-файлу
        String sequence = readSequenceFromFasta(fastaFilePath); // викликає readSequenceFromFastaметод для читання послідовності з файлу FASTA.

        // Розмір тестового набору маркерів
        int[] markerSetSizes = {100, 1000, 10000};  // Розміри набору тестових маркерів: масив, що представляє різні розміри для набору тестових маркерів.

        for (int setSize : markerSetSizes) {
            System.out.println("\nРозмір тестового набору маркерів: " + setSize);   // Перегляд розмірів набору маркерів: перебирає різні розміри набору маркерів.

            // Відібрані маркери
            List<String> selectedMarkers = markers.subList(0, setSize);  // створює підсписок маркерів на основі поточного розміру.
```

Рисунок 1 – Перша частина коду на мові програмування Java

Далі представлений код послідовної та паралельної реалізація алгоритму пошуку маркерів методом Бойера-Мура.



```
        // Послідовний пошук маркерів
        System.out.println("Послідовний пошук:");          // виконує послідовний пошук за допомогою алгоритму Боєра-Мура та вимірює витрачений час.
        long startTimeSeq = System.currentTimeMillis();
        for (String marker : selectedMarkers) {
            int index = boyerMooreSearch(sequence, marker);
            if (index != -1) {
                //System.out.println("Маркер '" + marker + "' знайдено за індексом " + index);
            } else {
                //System.out.println("Маркер '" + marker + "' не знайдено");
            }
        }
        long endTimeSeq = System.currentTimeMillis();
        double sequentialTime = millisecondsToMinutes(endTimeSeq - startTimeSeq);
        System.out.println("Час виконання послідовного пошуку: " + sequentialTime + " хвилин");

        // Паралельний пошук маркерів
        System.out.println("\nПараллельний пошук:");         // виконує паралельний пошук маркерів за допомогою ForkJoinPool і вимірює витрачений час.
        ForkJoinPool forkJoinPool = new ForkJoinPool();
        long startTimeParallel = System.currentTimeMillis();
        ParallelMarkerSearchTask searchTask = new ParallelMarkerSearchTask(sequence, selectedMarkers,  start: 0, selectedMarkers.size());
        List<Integer> results = forkJoinPool.invoke(searchTask);
        long endTimeParallel = System.currentTimeMillis();
        double parallelTime = millisecondsToMinutes(endTimeParallel - startTimeParallel);

        // Вивід результатів паралельного пошуку
        for (int i = 0; i < selectedMarkers.size(); i++) {      // друкує результати паралельного пошуку.
            if (results.get(i) != -1) {
                //System.out.println("Маркер '" + selectedMarkers.get(i) + "' знайдено за індексом " + results.get(i));
            } else {
                //System.out.println("Маркер '" + selectedMarkers.get(i) + "' не знайдено");
            }
        }

        System.out.println("Час виконання паралельного пошуку: " + parallelTime + " хвилин"); // друкує час, витрачений на паралельний пошук.

        // Вивід прискорення та ефективності
        int numProcessors = Runtime.getRuntime().availableProcessors();
        System.out.println("Число процесорів: " + numProcessors);    // друкує кількість процесорів, прискорення та ефективність для кожного розміру набору маркерів
        System.out.println("Прискорення для " + setSize + " маркерів: " + (sequentialTime / parallelTime));
        System.out.println("Ефективність для " + setSize + " маркерів: " + ((sequentialTime / parallelTime) / numProcessors));
    }
```

Рисунок 2 – Частина коду послідовної та паралельної реалізації алгоритму Бойєра-Мура

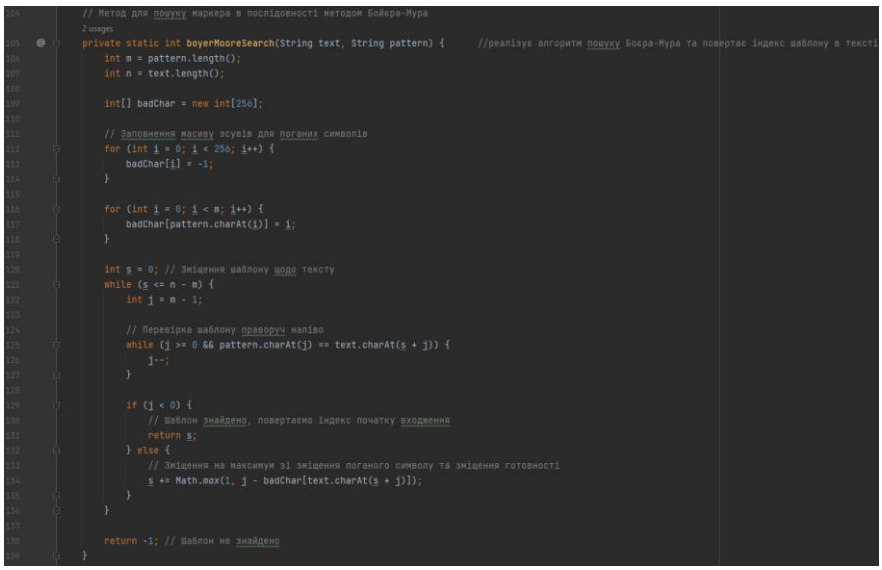
Метод boyerMooreSearch реалізує алгоритм Бойера-Мура для пошуку маркера у послідовності. ParallelMarkerSearchTask – клас, який поділяє завдання на підзавдання та поєднує результати. Розпаралелювання відбувається в методі ParallelMarkerSearchTask.compute, який є рекурсивним завданням, що використовується спільно з ForkJoinPool. Цей метод вирішує, чи слід шукати маркерів послідовно чи ділити завдання на підзавдання для паралельного виконання.

Процес розпаралелювання виглядає так:

1. Якщо кількість маркерів невелика (менша або дорівнює 1), то завдання виконується послідовно для кожного маркера в діапазоні.

2. Якщо маркерів багато, то завдання поділяється на два підзавдання, кожен з яких обробляє свою частину маркерів. Це робиться рекурсивно доти, доки кожна підзадача не обробляє окремий маркер.

Таким чином, паралельне виконання полягає в тому, щоб одночасно обробляти різні маркери у кількох потоках. Кожен потік відповідає за пошук маркера у своїй частині послідовності.



```
// Метод для пошуку маркера в послідовності методом Бойєра-Мура
2 usages
private static int boyerMooreSearch(String text, String pattern) {     //реалізує алгоритм пошуку Боєра-Мура та повертає індекс шаблону в тексті
    int m = pattern.length();
    int n = text.length();

    int[] badChar = new int[256];

    // Заповнення масиву зсувів для поганих символів
    for (int i = 0; i < 256; i++) {
        badChar[i] = -1;
    }

    for (int i = 0; i < m; i++) {
        badChar[pattern.charAt(i)] = i;
    }

    int s = 0; // Зміщення шаблону щодо тексту
    while (s <= n - m) {
        int j = m - 1;

        // Перевірка шаблону праворуч наліво
        while (j >= 0 && pattern.charAt(j) == text.charAt(s + j)) {
            j--;
        }

        if (j < 0) {
            // Шаблон знайдено, повертаємо індекс початку входження
            return s;
        } else {
            // Зміщення на максимум зі зміщення поганого символу та зміщення готовності
            s += Math.max(1, j - badChar[text.charAt(s + j)]);
        }
    }

    return -1; // Шаблон не знайдено
}
```

Рисунок 3 – Частина коду реалізації алгоритму Бойєра-Мура на Java

```
private static class ParallelMarkerSearchTask extends RecursiveTask<List<Integer>> { // визначає клас для пошуку паралельних маркерів за допомогою ForkJoinPool
    4 usages
    private final String sequence;
    4 usages
    private final List<String> markers;
    5 usages
    private final int start;
    5 usages
    private final int end;

    3 usages
    public ParallelMarkerSearchTask(String sequence, List<String> markers, int start, int end) {
        this.sequence = sequence;
        this.markers = markers;
        this.start = start;
        this.end = end;
    }
```

Рисунок 4 – Частина коду реалізації алгоритму Бойєра-Мура на Java

Результати виконання програми на Java зображені на рисунку 5:

```
C:\Users\Myroniuk\.jdks\openjdk-19.0.1\bin\java.exe "-javaagent:C

Розмір тестового набору маркерів: 100
Послідовний пошук:
Час виконання послідовного пошуку: 0.24978333333333333 хвилин

Параллельний пошук:
Час виконання паралельного пошуку: 0.06583333333333333 хвилин
Число процесорів: 4
Прискорення для 100 маркерів: 3.794177215189874
Ефективність для 100 маркерів: 0.9485443037974685

Розмір тестового набору маркерів: 1000
Послідовний пошук:
Час виконання послідовного пошуку: 2.4570666666666665 хвилин

Параллельний пошук:
Час виконання паралельного пошуку: 0.6400333333333333 хвилин
Число процесорів: 4
Прискорення для 1000 маркерів: 3.838966720483308
Ефективність для 1000 маркерів: 0.959741680120827

Розмір тестового набору маркерів: 10000
Послідовний пошук:
Час виконання послідовного пошуку: 24.6543 хвилин

Параллельний пошук:
Час виконання паралельного пошуку: 7.224366666666667 хвилин
Число процесорів: 4
Прискорення для 10000 маркерів: 3.412659010478427
Ефективність для 10000 маркерів: 0.8531647526196068

Process finished with exit code 0
```

Рисунок 5 – Результати роботи програми на Java

Реалізація алгоритму Бойєра-Мура на мові програмування Python.

Читання маркерів виконується за допомогою функції `read_markers_from_csv`. Послідовність читається з FASTA-файлу за допомогою функції `read_sequence_from_fasta.`

Паралельний пошук виконується за допомогою функції `parallel_marker_search`, яка розбиває набір маркерів на частини і запускає обробку кожної частини в окремих процесах за допомогою `ProcessPoolExecutor.`

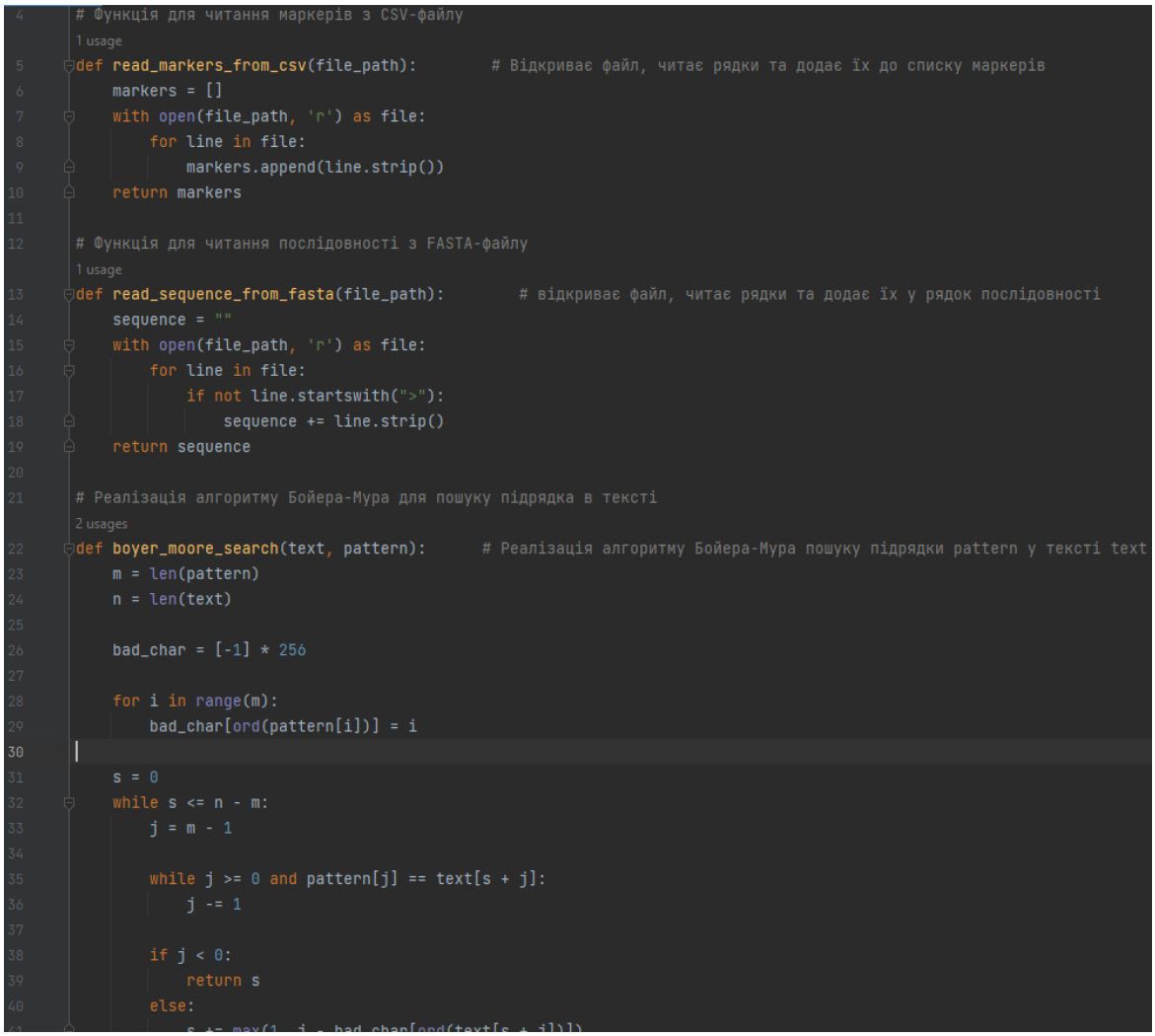
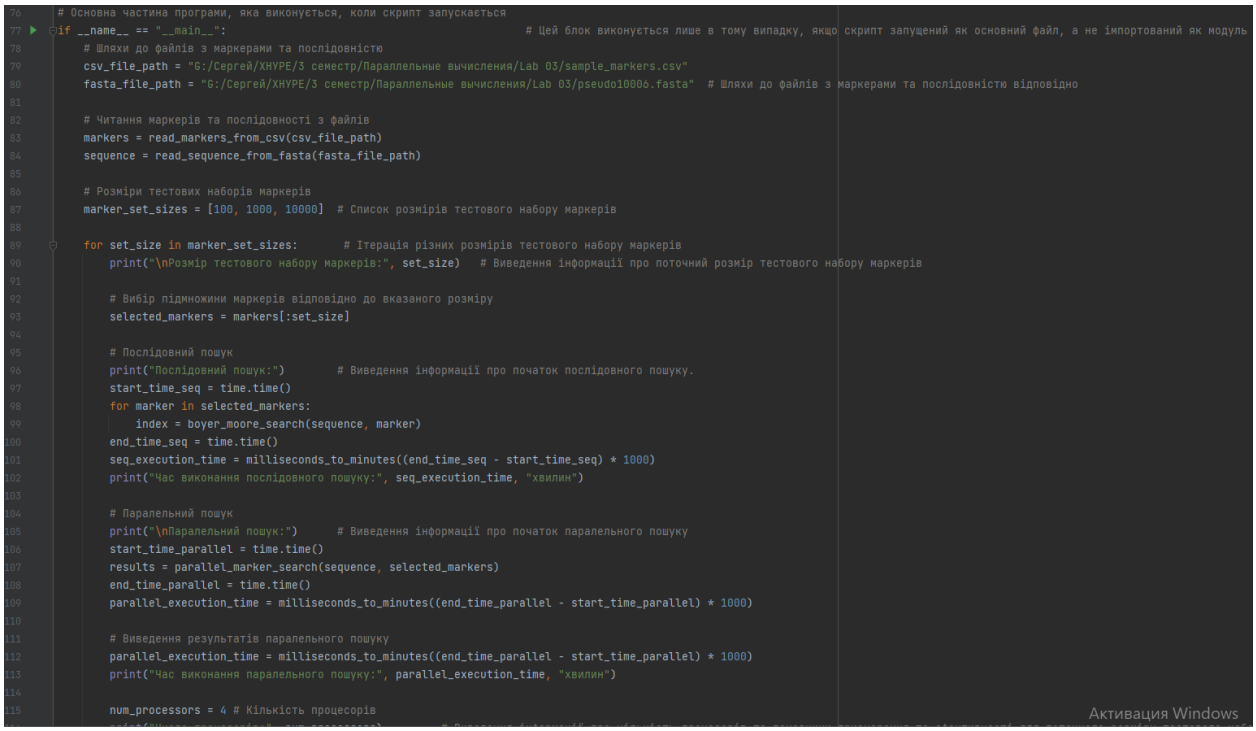


Рисунок 6 – Частина коду алгоритму на мові програмування Python

```
            else:
                s += max(1, j - bad_char[ord(text[s + j])])

        return -1

    # Функція для паралельного пошуку маркерів в послідовності
    1 usage
    def parallel_marker_search(sequence, markers, batch_size=10):       # Розбиває маркери на частини (marker_chunks), запускає процеси для обробки кожної частини та збирає результати
        results = []

        with concurrent.futures.ProcessPoolExecutor() as executor:
            marker_chunks = [markers[i:i + batch_size] for i in range(0, len(markers), batch_size)]
            futures = []

            for chunk in marker_chunks:
                future = executor.submit(process_marker_chunk, sequence, chunk)
                futures.append(future)

            for future in concurrent.futures.as_completed(futures):
                results.extend(future.result())

        return results

    # Функція для обробки частини маркерів в паралельному пошуку
    1 usage
    def process_marker_chunk(sequence, marker_chunk):
        chunk_results = []

        for marker in marker_chunk:                          # Функція обробки частини маркерів. Запускає пошук кожного маркера у послідовності та повертає результати
            index = boyer_moore_search(sequence, marker)
            chunk_results.append(index)

        return chunk_results
```

Рисунок 7 – Частина коду алгоритму на мові програмування Python



```
# Основна частина програми, яка виконується, коли скрипт запускається
if __name__ == "__main__":                                  # Цей блок виконується лише в тому випадку, якщо скрипт запущений як основний файл, а не імпортований як модуль
    # Шляхи до файлів з маркерами та послідовністю
    csv_file_path = "G:/Сергей/ХНУРЕ/3 семестр/Параллельные вычисления/Lab 03/sample_markers.csv"
    fasta_file_path = "G:/Сергей/ХНУРЕ/3 семестр/Параллельные вычисления/Lab 03/pseudo10006.fasta"  # Шляхи до файлів з маркерами та послідовністю відповідно

    # Читання маркерів та послідовності з файлів
    markers = read_markers_from_csv(csv_file_path)
    sequence = read_sequence_from_fasta(fasta_file_path)

    # Розміри тестових наборів маркерів
    marker_set_sizes = [100, 1000, 10000]  # Список розмірів тестового набору маркерів

    for set_size in marker_set_sizes:       # Ітерація різних розмірів тестового набору маркерів
        print("\nРозмір тестового набору маркерів:", set_size)   # Виведення інформації про поточний розмір тестового набору маркерів

        # Вибір підмножини маркерів відповідно до вказаного розміру
        selected_markers = markers[:set_size]

        # Послідовний пошук
        print("Послідовний пошук:")        # Виведення інформації про початок послідовного пошуку.
        start_time_seq = time.time()
        for marker in selected_markers:
            index = boyer_moore_search(sequence, marker)
        end_time_seq = time.time()
        seq_execution_time = milliseconds_to_minutes((end_time_seq - start_time_seq) * 1000)
        print("Час виконання послідовного пошуку:", seq_execution_time, "хвилин")

        # Паралельний пошук
        print("\nПаралельний пошук:")      # Виведення інформації про початок паралельного пошуку
        start_time_parallel = time.time()
        results = parallel_marker_search(sequence, selected_markers)
        end_time_parallel = time.time()
        parallel_execution_time = milliseconds_to_minutes((end_time_parallel - start_time_parallel) * 1000)

        # Виведення результатів паралельного пошуку
        parallel_execution_time = milliseconds_to_minutes((end_time_parallel - start_time_parallel) * 1000)
        print("Час виконання паралельного пошуку:", parallel_execution_time, "хвилин")

        num_processors = 4 # Кількість процесорів
```


Рисунок 8 – Частина коду алгоритму на мові програмування Python

Результати виконання програми представлені на рисунку 9.

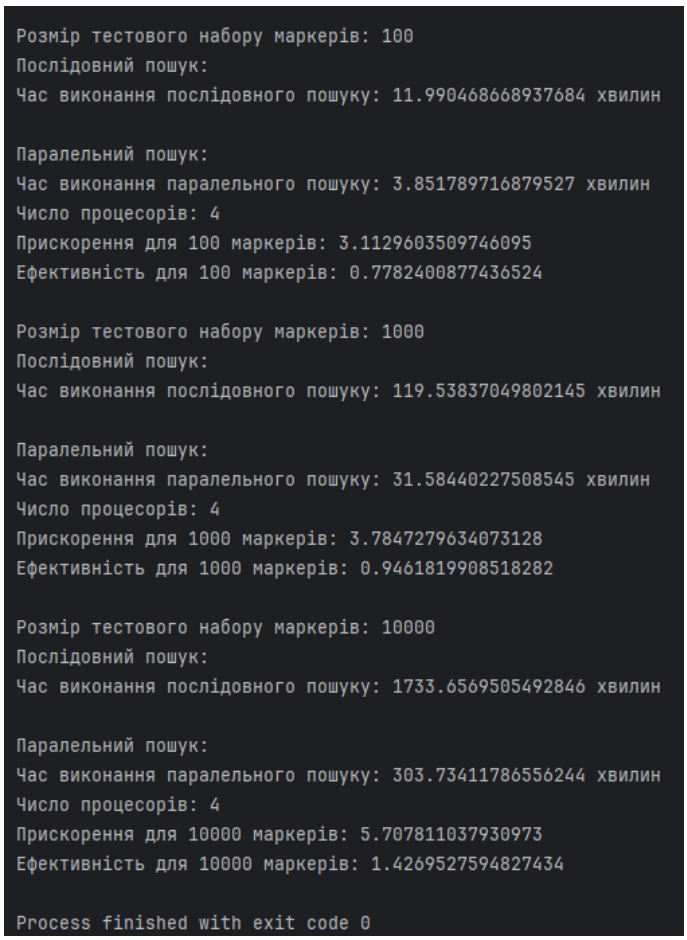


Рисунок 9 – Результати роботи програми на мові програмування Python

Таблиця 1 – Порівняння результатів послідовної та паралельної реалізацій алгоритму Бойєра-Мура на Java і Python

	Java						Python					
	послідовна реалізація алгоритму			паралельна реалізація алгоритму			послідовна реалізація алгоритму			паралельна реалізація алгоритму		
	100	1000	10000	100	1000	10000	100	1000	10000	100	1000	10000
Час виконання, хв.	0,25	2,45	24,65	0,06	0,64	7,22	11,99	111,5	1733,6	3,85	31,58	303,3
Прискорення ($S = \frac{\text{Тпослід.}}{\text{Тпарал.}}$)	-	-	-	3,79	3,83	3,41	-	-	-	3,11	3,78	5,71
Ефективність ($E = \frac{S}{\text{число потоків}}$)	-	-	-	0,94	0,95	0,85	-	-	-	0,79	0,95	1,43

На рисунку 10 зображений графік порівнянням часу виконання програми на Java і на Python.

Рисунок 10 – Графік порівняння часу виконання програми на Java і на Python

Висновки: в ході роботи засвоїли навички створювати і аналізувати ітераційні паралельні алгоритми виявлення підпослідовностей в однорідних текстових безперервних потоках даних для різних програмних середовищ.

За результатами виконання роботи можна побачити, що виконання програми на мові Java значно ефективніше порівняно з виконанням програми на Python.

Ось деякі фактори, які слід враховувати:

1. Python має глобальне блокування інтерпретатора (GIL), яке дозволяє лише одному потоку виконувати байт-код Python одночасно в одному процесі. Це може обмежити паралельне виконання потоків та вплинути на продуктивність завдань, пов'язаних із ЦП. Навпаки, Java зазвичай використовує власні потоки, та її модель паралелізму дозволяє краще використовувати кілька процесорів.

2. Модель паралелізму: Python код використовує concurrent.futures.ThreadPoolExecutor для паралельного виконання, але через GIL потоки можуть не досягти істинного паралелізму. Java, з іншого боку, використовує ForkJoinPool, який призначений для паралелізму і може краще використати кілька процесорів.

3. Відмінності у реалізації. Ефективність деяких операцій, таких як файлове виведення та маніпулювання рядками, може різнитися в різних реалізаціях Python і Java. Java може мати оптимізовані бібліотеки або виконувати певні завдання ефективніше, ніж еквівалентний код Python.

4. Рівні оптимізації: Java часто виграє від JIT-компіляції, коли байт-код перетворюється на машинний код під час виконання, що дозволяє проводити потенційну оптимізацію. Python — це мова, що інтерпретується, інтерпретатор CPython за умовчанням може не виконувати стільки оптимізацій під час виконання.

5. Накладні витрати на обробку потоків. Накладні витрати на створення та керування потоками в Python можуть бути вищими, ніж у Java, що впливає на продуктивність паралельного виконання.

6. Управління пам'яттю. Стратегії керування пам'яттю можуть відрізнятися в Python і Java, і це може вплинути на продуктивність операцій, що інтенсивно використовують пам'ять.

Відповіді на контрольні запитання:

1. Сформулювати основні проблеми пов’язані з пошуком підпослідовностей у текстових даних.

Існують такі основні недоліки:

1) Значна об'ємність даних – опрацювання значної кількості текстових даних може бути витратним з точки зору часу та ресурсів.

2) Ефективність алгоритмів – вибір оптимального алгоритму для пошуку підпослідовностей є складною задачею, особливо при різноманітності типів даних та шаблонів.

3) Неспецифічні шаблони – деякі методи можуть бути неефективними при виявленні непередбачуваних та неявних шаблонів у тексті.

4) Обробка шуму – наявність шуму або випадкових варіацій у тексті може призвести до помилкових результатів при визначенні підпослідовностей.

5) Специфічні вимоги до даних – деякі алгоритми можуть потребувати конкретної структури даних або передпопередню обробку, що ускладнює їх використання в різних контекстах.

6) Використання пам'яті та ресурсів – ефективність алгоритмів пошуку підпослідовностей може залежати від доступних обсягів пам'яті та обчислювальних ресурсів.

7) Робота з текстом мовами різної складності – різниця в мовній структурі та граматиці може ускладнювати виявлення підпослідовностей у текстах різних мов.

4. У чому полягає сутність методу Рабіна-Карпа? Які його основні недоліки та переваги?

Алгоритм Рабіна-Карпа не використовує зсуви чи кінцеві автомати. Його сутність полягає в заміні рядкових змінних значеннями хещ-функції. Таким чином відбувається перехід від аналізу текстової інформації до аналізу числової.

Однак з цим підходом пов’язані дві наступні проблеми.

Перша полягає в тому, що, оскільки існує дуже багато різних рядків, між значеннями їхніх хеш-функцій може виникнути колізія. У таких випадках необхідно посимвольно перевіряти збіг самих підрядків, що може нівілювати виграш від переходу до числових змінних.

Друга проблема пов’язана з часом виконання хеш-функції. У наївному випадку виграш у швидкодії запропонованого підходу нівелюватиметься.

Враховуючи зазначені вище проблеми, Рабін та Карп запропонували використовувати поліноміальну хеш-функцію, визначену наступним чином:

$$hash(p[1..m]) = \left(\sum_{i=1}^{m} p[i]x^{m-i}\right) mod\ q,$$

де – деяке просте число, – число від 0 до , – довжина підпослідовності.

При цьому для ефективності хеш схожих підпослідовностей вираховується один з одного. Окрім безпосереднього використання хешу замість рядків, алгоритм не відрізняється від наївного.

Алгоритм Рабіна-Карпа особливо корисний завдяки своїй здатності виконувати зіставлення із зразком у середньому за лінійний час. Така ефективність досягається за рахунок використання ковзної хеш-функції, яка дозволяє обчислювати хеш-значення підрядка за постійний час. Це поєднується зі спостереженням, що якщо два рядки хешують те саме значення, вони можуть бути рівні, що дозволяє алгоритму швидко порівнювати підрядки.