Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"
Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт

з лабораторної роботи № 1 з дисципліни
«Проектування алгоритмів»

„ Проектування і аналіз алгоритмів зовнішнього сортування”

Виконав(ла) *ІП-24 Ротань Олександр Євгенович*
(шифр, прізвище, ім'я, по батькові)

Перевірив *Ахаладзе І. Е.*
(прізвище, ім'я, по батькові)

Київ 2023

1. МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи – вивчити основні алгоритми зовнішнього сортування та способи їх модифікації, оцінити поріг їх ефективності.

2. ЗАВДАННЯ

Згідно варіанту (таблиця 2.1), розробити та записати алгоритм зовнішнього сортування за допомогою псевдокоду (чи іншого способу за вибором).

Виконати програмну реалізацію алгоритму на будь-якій мові програмування та відсортувати випадковим чином згенерований масив цілих чисел, що зберігається у файлі (розмір файлу має бути не менше 10 Мб, можна значно більше).

Здійснити модифікацію програми і відсортувати випадковим чином згенерований масив цілих чисел, що зберігається у файлі розміром не менше ніж двократний обсяг ОП вашого ПК. Досягти швидкості сортування з розрахунку 1Гб на 3хв. або менше. Достатньо штучно обмежити доступну ОП, для уникнення багатогодинних сортувань (наприклад використовуючи віртуальну машину).

Рекомендується попередньо впорядкувати серії елементів довжиною, що займає не менше 100Мб або використати інші підходи для пришвидшення процесу сортування.

Зробити узагальнений висновок з лабораторної роботи, у якому порівнятибазову та модифіковану програми. У висновку деталізувати, які саме модифікації було виконано і який ефект вони дали.

Варіант 19 - Збалансоване багатошляхове злиття

3. ВИКОНАННЯ

3.1. Псевдокод алгоритму

```
class ExternalFileSortExecutor implements AlgorithmExecutor {

   function mergeChunks(outputFileName, inputFilePath, chunksAmount):
      outputFile = create RandomAccessFile(outputFileName, "rw")
      outputBuffer = create IntBuffer
      channel = outputFile.getChannel()
      outputBuffer = map channel to FileChannel.MapMode.READ_WRITE

      chunkBuffers = create IntBuffer array of size chunksAmount
      for i from 0 to chunksAmount-1 do
         channel = open FileChannel for "chunk" + i + ".tmp.bin"
         chunkBuffers[i] = map channel to
FileChannel.MapMode.READ_ONLY
      end for

      mergeBuffers(chunksAmount, chunkBuffers, outputBuffer)

   end function

   function mergeBuffers(chunksAmount, chunkBuffers, outputBuffer):
      peeks = create PriorityQueue of SimpleImmutableEntry<Integer,
Integer> with custom comparator
      for j from 0 to chunksAmount-1 do
         peeks.add(new SimpleImmutableEntry(j, chunkBuffers[j].get()))
      end for

      last = 0
      while not peeks.isEmpty() do
```

```
            min = peeks.poll()
            last = min.key
            outputBuffer.put(min.value)

            if chunkBuffers[last].hasRemaining() then
                peeks.add(new SimpleImmutableEntry(last,
chunkBuffers[last].get()))
            end if
        end while

    end function

    function splitInChunks(inputFilePath, chunksAmount, executor,
sortCompletionTracker):
        channel = open FileChannel for inputFilePath
        mappedBuffer = map channel to FileChannel.MapMode.READ_ONLY

        for i from 0 to chunksAmount-1 do
            chunkSize = mappedBuffer.capacity() / chunksAmount

            if i == chunksAmount - 1 then
                chunkSize += mappedBuffer.capacity() % chunksAmount
            end if

            chunk = create IntBuffer
            chunk = slice mappedBuffer
            chunk.limit(chunkSize)
            mappedBuffer.position(mappedBuffer.position() + chunkSize)
            executor.submit(new ChunkSortThread(sortCompletionTracker,
chunk, chunkSize, i))
```

```
            end for

            sortCompletionTracker.arriveAndAwaitAdvance()

        end function

        function execute(params):
            start = current time in milliseconds
            inputFileName = params.get("inputFileName")
            outputFileName = params.get("outputFileName")
            inputFilePath = create Path from inputFileName
            chunksAmount = 3
            sortCompletionTracker = create Phaser with initial count of
    chunksAmount + 1

            executor = create virtual thread per task ExecutorService
            splitInChunks(inputFilePath, chunksAmount, executor,
    sortCompletionTracker)
            mergeChunks(outputFileName, inputFilePath, chunksAmount)
            sortCompletionTracker.arriveAndAwaitAdvance()

            end = current time in milliseconds
            print "Completed"
            print "Time: " + (end - start) + " ms"
        end function
}
    class ChunkSortThread implements Runnable {

        // Constructor
        function ChunkSortThread(sortCompletionTracker, inputBuffer,
```

```
chunkSize, chunkNumber):
        this.sortCompletionTracker = sortCompletionTracker
        this.inputBuffer = inputBuffer
        this.chunkSize = chunkSize
        this.chunkNumber = chunkNumber
    end function

    // Static method
    function convertToBytes(chunk, bytes):
        for i from 0 to chunk.length - 1 do
            bytes[i * 4] = (byte)(chunk[i] >> 24)
            bytes[i * 4 + 1] = (byte)(chunk[i] >> 16)
            bytes[i * 4 + 2] = (byte)(chunk[i] >> 8)
            bytes[i * 4 + 3] = (byte)chunk[i]
        end for
    end function

    // Override run method
    function run():
        fileName = "chunk" + chunkNumber + ".tmp.bin

        chunk = readChunk()
        sort chunk
        bytes = create byte array of size chunk.length * 4
        convertToBytes(chunk, bytes)

        file = create File with name fileName
        file.createNewFile() // Ignore result of method call
        outputStream = create FileOutputStream for file
        outputStream.write(bytes, 0, bytes.length)
```

```
            outputStream.close()


            sortCompletionTracker.arriveAndAwaitAdvance()
            sortCompletionTracker.arriveAndAwaitAdvance()
            Files.delete(Path.of(fileName))
        end function


        function readChunk():
            chunk = create int array of size chunkSize
            inputBuffer.get(chunk)
            return chunk
        end function
}
```

3.2. Код алгоритму

```
package org.example.algo.externalsort;

import org.example.algo.AlgorithmExecutor;

import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.IntBuffer;
import java.nio.channels.FileChannel;
import java.nio.file.Path;
import java.util.AbstractMap;
import java.util.Comparator;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Phaser;

public class ExternalFileSortExecutor implements AlgorithmExecutor {
    private void mergeChunks(String outputFileName, Path inputFilePath, int
chunksAmount) {
        try (var outputFile = new RandomAccessFile(outputFileName, "rw")) {
            IntBuffer outputBuffer;
            try (var channel = outputFile.getChannel()) {
                outputBuffer                                                =
```

```
channel.map(FileChannel.MapMode.READ_WRITE, 0,
                    inputFilePath.toFile().length()).asIntBuffer();
          }
          var chunkBuffers = new IntBuffer[chunksAmount];
          for (var i = 0; i < chunksAmount; i++) {
            try (var channel = FileChannel.open(Path.of("chunk" + i +
".tmp.bin"))) {
                chunkBuffers[i]                                          =
channel.map(FileChannel.MapMode.READ_ONLY, 0,
                      channel.size()).asIntBuffer();
            }
          }
          mergeBuffers(chunksAmount, chunkBuffers, outputBuffer);
        } catch (IOException e) {
          throw new RuntimeException(e);
        }
    }

    private void mergeBuffers(int chunksAmount, IntBuffer[] chunkBuffers,
IntBuffer outputBuffer) {
        var                peeks                =                new
PriorityQueue<AbstractMap.SimpleImmutableEntry<Integer,
Integer>>(chunksAmount,

Comparator.comparingInt(AbstractMap.SimpleImmutableEntry::getValue));
        for (int j = 0; j < chunksAmount; j++) {
          peeks.add(new                AbstractMap.SimpleImmutableEntry<>(j,
chunkBuffers[j].get()));
        }
        int last;
        while (!peeks.isEmpty()) {
          var min = peeks.poll();
          last = min.getKey();
          outputBuffer.put(min.getValue());
          if (chunkBuffers[last].hasRemaining()) {
            peeks.add(new          AbstractMap.SimpleImmutableEntry<>(last,
chunkBuffers[last].get()));
          }
        }
    }

    private void splitInChunks(Path inputFilePath,
                               int chunksAmount,
                               ExecutorService executor,
                               Phaser sortCompletionTracker) {
```

```
        IntBuffer mappedBuffer;
        try (var channel = FileChannel.open(inputFilePath)) {
            mappedBuffer =
    channel.map(FileChannel.MapMode.READ_ONLY, 0,
                    inputFilePath.toFile().length()).asIntBuffer();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        for (var i = 0; i < chunksAmount; i++) {
            var chunkSize = (mappedBuffer.capacity() / chunksAmount +
                    (i == chunksAmount - 1 ? mappedBuffer.capacity() %
    chunksAmount : 0));
            var chunk = mappedBuffer.slice();
            chunk.limit(chunkSize);
            mappedBuffer.position(mappedBuffer.position() + chunkSize);
            executor.submit(new ChunkSortThread(sortCompletionTracker,
    chunk, chunkSize, i));
        }
        sortCompletionTracker.arriveAndAwaitAdvance();
    }

    @Override
    public void execute(Map<String, Object> params) {
        var start = System.currentTimeMillis();
        var inputFileName = (String) params.get("inputFileName");
        var outputFileName = (String) params.get("outputFileName");
        var inputFilePath = Path.of(inputFileName);
        var chunksAmount = 3;
        var sortCompletionTracker = new Phaser(chunksAmount + 1);

        try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
            splitInChunks(inputFilePath, chunksAmount, executor,
    sortCompletionTracker);
            mergeChunks(outputFileName, inputFilePath, chunksAmount);
            sortCompletionTracker.arriveAndAwaitAdvance();
            var end = System.currentTimeMillis();
            System.out.println("Completed");
            System.out.println("Time: " + (end - start) + " ms");
        }
    }
}
```

```
    package org.example.algo.externalsort;

    import java.io.File;
```

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.IntBuffer;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.Arrays;
import java.util.concurrent.Phaser;

public class ChunkSortThread implements Runnable {
    private final Phaser sortCompletionTracker;

    private final IntBuffer inputBuffer;

    private final int chunkSize;

    private final int chunkNumber;

    public ChunkSortThread(Phaser sortCompletionTracker,
                           IntBuffer inputBuffer,
                           int chunkSize,
                           int chunkNumber) {
        this.sortCompletionTracker = sortCompletionTracker;
        this.inputBuffer = inputBuffer;
        this.chunkSize = chunkSize;
        this.chunkNumber = chunkNumber;
    }

    private static void convertToBytes(int[] chunk, byte[] bytes) {
        for (int i = 0; i < chunk.length; i++) {
            bytes[i * 4] = (byte) (chunk[i] >> 24);
            bytes[i * 4 + 1] = (byte) (chunk[i] >> 16);
            bytes[i * 4 + 2] = (byte) (chunk[i] >> 8);
            bytes[i * 4 + 3] = (byte) chunk[i];
        }
    }

    @Override
    public void run() {
        String fileName = "chunk" + chunkNumber + ".tmp.bin";
        try {
            var chunk = readChunk();

            Arrays.sort(chunk);

            var bytes = new byte[chunk.length * 4];
```

```
                convertToBytes(chunk, bytes);

                var file = new File(fileName);
                //noinspection ResultOfMethodCallIgnored
                file.createNewFile();
                try (var outputStream = new FileOutputStream(file)) {
                    outputStream.write(bytes, 0, bytes.length);
                }
            } catch (InterruptedException | IOException e) {
                throw new RuntimeException(e);
            }
            sortCompletionTracker.arriveAndAwaitAdvance();
            sortCompletionTracker.arriveAndAwaitAdvance();
            try {
                Files.delete(Path.of(fileName));
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }

        private int[] readChunk() throws InterruptedException {
            var chunk = new int[chunkSize];
            inputBuffer.get(chunk);
            return chunk;
        }
}
```

ВИСНОВОК

У ході виконання лабораторної роботи я реалізував алгоритм зовнішнього багатошляхового сортування та оптимізував його за допомогою паралельного сортування шляхом використання віртуальних потоків а також відображення буферу в пам’ять що майже звільнило програму від найповільнішого місця – операцій з диском.