**Министерство науки высшего образования Российской Федерации**

**федеральное государственное автономное образовательное учреждение высшего образования**

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»**

**(Университет ИТМО)**

**Факультет программной инженерии и компьютерной техники**

ЛАБОРАТОРНАЯ РАБОТА №2

По курсу «Реактивная Java»

Предметная область «Матчи, турниры, команды»

Выполнили:

Голоскок Дмитрий Сергеевич P4119

Симовин Кирилл Константинович P4116

Преподаватель:

Гаврилов Антон Валерьевич

Санкт-Петербург

2024

Задание

1. В один из методов, использовавшийся для сбора статистики, добавить возможность задать задержку, имитирующую задержку получения результата, например, из базы данных. К примеру, был метод getName(), в который нужно добавить параметр getName(long delay).

2. Заменить последовательный стрим, собирающий статистику из лабораторной №1, на параллельный. Измерить производительность для разного количества элементов с дополнительной задержкой и без задержки. Для случаев с задержкой и без найти количество элементов, при котором сбор статистики последовательным и параллельным стримами дают одинаковую скорость выполнения.

3. Оптимизировать параллельный сбор статистики, реализовав собственный ForkJoinPool или Spliterator. Измерить производительность своего варианта.

4. Измерения производительности выполнять с помощью фрейворка JMH.

Ход выполнения работы

В ходе исследования времени работы программ здесь и далее проводилось усреднение по 12 запускам.

В таблице 1 представлено время выполнения программ в зависимости от размера обрабатываемых коллекций и задержки. Время задержки устанавливалось равное 5 секундам.

Таблица 1 – Сравнительная таблица методов обработки

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N | Время выполнения программ, с | | | | | |
| Цикл | | Стандартный коллектор | | Собственный коллектор | |
|  | С задержкой | Без задержки | С задержкой | Без задержки | С задержкой | Без задержки |
| 5000 | 0,02317 | 0,01417 | 0,01467 | 0,00717 | 0,01317 | 0,00450 |
| 50000 | 0,03242 | 0,00908 | 0,01917 | 0,00767 | 0,01742 | 0,00800 |
| 250000 | 0,03625 | 0,02783 | 0,08392 | 0,05000 | 0,07942 | 0,05375 |

Таблица 2 – Общее количество элементов для программ с задержкой и без

|  |  |  |
| --- | --- | --- |
| Тип | Стандартный коллектор | Собственный коллектор |
| С задержкой | 1562250 | 1562500 |
| Без задержки | 1830500 | 1830500 |

Таблица 3 – Сравнительная таблица методов обработки с использованием ForkPoolJoin

|  |  |  |
| --- | --- | --- |
| N | Время выполнения программы, с | |
| Без задержки | С задержкой |
| 5000 | 0,01442 | 0,03981 |
| 50000 | 0,092742 | 0,12812 |
| 250000 | 0,10319 | 0, 20714 |

Как можно заметить, программа с использованием цикла работает быстрее в двух из трех случаев – не считая теста на 5000 тысячах элементах.

На рисунке 1 приведен график зависимости времени работы программ от размера коллекций.

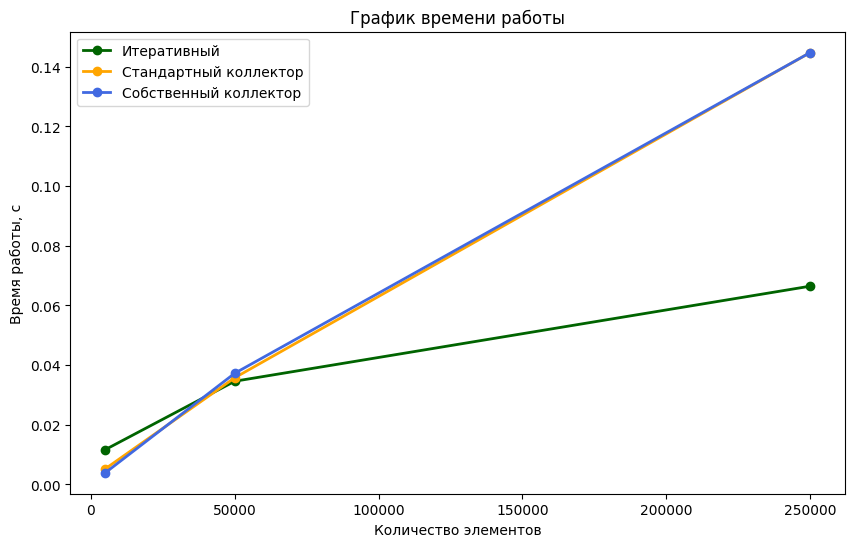


Рисунок 1 – График времени работы программ

Заключение

Цель лабораторной работы – сравнить скорость работы трех решений для расчета статистических данных. Показано, что на предоставленном размере коллекций итеративный способ является более предпочтительным.

Однако, для 5000 элементов итеративный способ является более медленным, по сравнению с коллекторами: 0,01167 секунд против 0,00525 секунд для стандартного коллектора и 0,00392 секунд с использованием собственного коллектора.

Для 50000 элементов решения с коллекторами уступили итеративному подходу: 0,03458 секунд для итеративного против 0,03575 секунд для решения с использованием стандартного коллектора и 0,03725 секунд с использованием собственного.

Для 250000 элементов итеративный подход оказался более, чем в два раза быстрее: 0,06642 секунды против 0,14467 секунд для обоих решений с коллектором.

В ходе выполнения лабораторной работы были получены навыки по работе с коллекторами и реализации собственных.

Переделать:

-уменьшение задержки до 0.05 с

- сравнение последовательной и параллельной работы

- JoinPool оптимизировать разбиение на группы

- группировать элементы по одному признаку

- изменять кол-во элементов порядками

ПРИЛОЖЕНИЕ А  
Код Record-класса

import org.jetbrains.annotations.Contract;  
import org.jetbrains.annotations.NotNull;  
  
import java.time.LocalDate;

public record Tournament(String name, String place, LocalDate startDate, LocalDate endDate) {@Contract(pure = true)  
 @Override  
 public @NotNull String toString() {  
 return "Tournament{name=%s, place=%s, startDate=%s, endDate=%s}"  
 .formatted(this.name, this.place, this.startDate, this.endDate);  
 }  
}

ПРИЛОЖЕНИЕ Б  
Код класса с коллекцией

import lombok.AllArgsConstructor;  
import lombok.Data;   
  
import java.util.List;

@Data  
@AllArgsConstructor  
public class Team {private String name;  
private List<String> members;  
private int wins;  
private int loses;  
}

ПРИЛОЖЕНИЕ В  
Код главного класса

import lombok.AllArgsConstructor;  
import lombok.Data;  
import org.example.enums.MatchType;   
  
import java.time.LocalDateTime;

@Data  
@AllArgsConstructor  
public class Match {

private Team team1;

private Team team2;

private LocalDateTime startDateTime;

private LocalDateTime endDateTime;

private Tournament tournament;

private int scoreTeam1;

private int scoreTeam2;

private MatchType matchType;

private String map;  
}

ПРИЛОЖЕНИЕ Г  
Код перечисления

import lombok.AllArgsConstructor;  
import lombok.Getter;

@Getter  
@AllArgsConstructor  
public enum MatchType {

DEATHMATCH("Deathmatch"),

FLAG\_CAPTURE("Flag Capture"),

TEAM\_BATTLE("Team Battle");  
  
 private final String type;  
}

ПРИЛОЖЕНИЕ Д  
Код коллектора

import org.example.entity.Match;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Set;  
import java.util.function.BiConsumer;  
import java.util.function.BinaryOperator;  
import java.util.function.Function;  
import java.util.function.Supplier;  
import java.util.stream.Collector;

public class ReactiveCollector implements Collector<Match, List<Match>, List<Match>> {@Override  
 public Supplier<List<Match>> supplier() {  
 return ArrayList::new;  
 }

@Override  
 public BiConsumer<List<Match>, Match> accumulator() {  
 return List::add;  
 }

@Override  
 public BinaryOperator<List<Match>> combiner() {  
 return (list1, list2) -> {  
 list1.addAll(list2);  
 return list1;  
 };  
 }

@Override  
 public Function<List<Match>, List<Match>> finisher() {  
 return Function.*identity*();  
 }  
@Override  
 public Set<Characteristics> characteristics() {  
 return Set.*of*();  
 }  
}