Laboratorium VIII

***Asynchroniczne wykonanie zadań w puli***

***wątków przy użyciu wzorców Executor i Future***

*Dominik Marek*

*25 listopada 2024*

Obraz zawierający tekst, Czcionka, logo, Grafika

Opis wygenerowany automatycznie

***1.Zadania***

**Interfejs java.util.concurrent.Executor i java.util.concurrent.ExecutorService**

1. Executor służy do asynchronicznego wykonania zadań typu Runnable. Zamiast tworzyć osobne wątki:
2. new Thread(new(RunnableTask())).start()

można użyć metody execute():

executor.execute(new RunnableTask1());

executor.execute(new RunnableTask2());

1. Metoda submit() w interfejsie ExecutorService działa podobnie do execute(), ale przyjmuje zadania implementujące interfejs Callable, które mogą zwrócić wartość (metoda run() jest typu void). Zadanie może implementować interfejs Future.
2. Implementacje:
   * newSingleThreadExecutor
   * newFixedThreadPool
   * newCachedThreadPool
   * newWorkStealingPool

**Zadania**

1. Proszę zaimplementować przy użyciu Executor i Future program wykonujący obliczanie zbioru Mandelbrota w puli wątków. Jako podstawę implementacji proszę wykorzystać [kod w Javie](http://rosettacode.org/wiki/Mandelbrot_set#Java).
2. Proszę przetestować szybkość działania programu w zależności od implementacji Executora i jego parametrów (np. liczba wątków w puli). Czas obliczeń można zwiększać manipulując parametrami problemu, np. liczbą iteracji (MAX\_ITER).

**2.Rozwiązania**

Poniższy kod w języku Java generuje i renderuje fraktal Mandelbrota, przy wykorzystaniu podejścia równoległego do obliczeń pikseli. Klasa PixelOperator implementuje interfejs Callable i odpowiada za obliczenie koloru dla pojedynczego piksela na podstawie równania Mandelbrota. Każda instancja tej klasy działa na określonym punkcie współrzędnych (x, y) w obrazie i zwraca wynik w formie obiektu PixelResult. Klasa Mandelbrot jest główną klasą programu i rozszerza JFrame, umożliwiając graficzne wyświetlenie wygenerowanego obrazu. Używa wątku ExecutorService z różnymi liczbami wątków (od 1 do 20), aby równolegle obliczać piksele obrazu. Wyniki obliczeń pikseli są pobierane za pomocą listy Future i zapisywane do obiektu BufferedImage, który przechowuje gotowy obraz. Metoda runSimulation odpowiada za podział pracy na równoległe zadania i koordynację procesu obliczeń oraz renderowania obrazu. Wyświetlenie obrazu odbywa się w metodzie paint, która rysuje wynikowy fraktal na ekranie. W pętli testowej w konstruktorze programu mierzony jest czas wykonania symulacji dla różnych liczb wątków, a wyniki są wyświetlane w konsoli. Dzięki temu można przeanalizować wpływ liczby wątków na czas generowania fraktala. Kod demonstruje równoległe przetwarzanie i wizualizację złożonych obliczeń matematycznych, przy czym wykorzystywany jest model programowania współbieżnego w Javie

import java.util.concurrent.Callable;  
  
public class PixelOperator implements Callable<PixelResult> {  
 private final int x, y;  
 private final double ZOOM;  
 private final int MAX\_ITER;  
  
  
 public PixelOperator(int x, int y , double ZOOM, int MAX\_ITER) {  
 this.x = x;  
 this.y = y;  
 this.ZOOM = ZOOM;  
 this.MAX\_ITER = MAX\_ITER;  
 }  
  
  
 @Override  
 public PixelResult call() {  
 double zx, cX, cY, zy;  
 zx = zy = 0;  
 cX = (x - 400) / ZOOM;  
 cY = (y - 300) / ZOOM;  
 int iter = MAX\_ITER;  
 while (zx \* zx + zy \* zy < 4 && iter > 0) {  
 double tmp = zx \* zx - zy \* zy + cX;  
 zy = 2.0 \* zx \* zy + cY;  
 zx = tmp;  
 iter--;  
 }  
 int color = iter | (iter << 8);  
 return new PixelResult(this.x, this.y, color);  
 }  
}

record PixelResult(int x, int y, int color) {  
}

import java.awt.Graphics;  
import java.awt.image.BufferedImage;  
import java.util.ArrayList;  
import java.util.List;  
import java.util.concurrent.\*;  
import javax.swing.JFrame;  
  
public class Mandelbrot extends JFrame {  
  
 private static final int *MAX\_ITER* = 3000;  
 private static final double *ZOOM* = 150;  
 private BufferedImage I;  
  
  
 public Mandelbrot() {  
 System.*out*.println("newFixedThreadPool:\n");  
 List<Integer> threadsNum = List.*of*(1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20);  
 List<Double> executionTime = new ArrayList<>();  
  
 for(int nThreads : threadsNum) {  
 ExecutorService executor = Executors.*newFixedThreadPool*(nThreads);  
 double startTime = System.*nanoTime*();  
 runSimulation(executor);  
 double endTime = System.*nanoTime*();  
 executionTime.add((endTime - startTime) / 1\_000\_000.0);  
 }  
 for(double execTime: executionTime) {  
 //System.out.println(execTime);  
 System.*out*.println("Number of threads: "+threadsNum.get(executionTime.indexOf(execTime)) + " time: "+ execTime);  
 }  
  
  
 }  
  
  
 public void runSimulation(ExecutorService executor) {  
  
 setBounds(100, 100, 800, 600);  
 setResizable(false);  
 setDefaultCloseOperation(*EXIT\_ON\_CLOSE*);  
 I = new BufferedImage(getWidth(), getHeight(), BufferedImage.*TYPE\_INT\_RGB*);  
  
 List<Future<PixelResult>> futures = new ArrayList<>();  
  
 for (int y = 0; y < getHeight(); y++) {  
 for (int x = 0; x < getWidth(); x++) {  
 Callable<PixelResult> task = new PixelOperator(x, y, *ZOOM*, *MAX\_ITER*);  
 futures.add(executor.submit(task));  
  
 }  
 }  
  
 for (Future<PixelResult> future : futures) {  
 try {  
 PixelResult result = future.get();  
 I.setRGB(result.x(), result.y(), result.color());  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
  
 executor.shutdown();  
 }  
  
  
 @Override  
 public void paint(Graphics g) {  
 g.drawImage(I, 0, 0, this);  
 }  
  
 public static void main(String[] args) {  
 new Mandelbrot().setVisible(true);  
 }  
}

|  |  |
| --- | --- |
| **Czas wykonania programu względem liczby wątków dla FIxedThreadExecutor** | |
| **liczba wątków** | **czas wykonania [ ms]** |
| 1 | 653,2196 |
| 2 | 427,1624 |
| 4 | 264,7698 |
| 6 | 210,6304 |
| 8 | 200,3586 |
| 10 | 200,1025 |
| 12 | 170,2929 |
| 14 | 189,5025 |
| 16 | 159,1747 |
| 18 | 208,6295 |
| 20 | 159,7834 |

Analizując powyższy wykres możemy zauważyć, że zwiększanie liczby wątków początkowo znacząco skraca czas wykonania programu, szczególnie przy przejściu z 1 do 4 wątków. Optymalny czas (ok. 159 ms) osiągnięto przy 12-16 wątkach, co odpowiada liczbie logicznych wątków procesora Intel i7-9750H (6 rdzeni, 12 wątków). Po przekroczeniu tej liczby czas wykonania zaczyna rosnąć, co wynika z narzutu związanego z zarządzaniem wątkami i przełączaniem kontekstu.

**Wnioski:**

Zastosowanie liczby wątków zbliżonej do faktycznej liczby wątków procesora zapewnia najlepszą wydajność, minimalizując czas obliczeń. Dodawanie większej liczby wątków nie poprawia efektywności, a może ją pogorszyć. Warto dostosowywać liczbę wątków do liczby logicznych rdzeni procesora dla optymalizacji programów równoległych.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Czas wykonania programu względem liczby iteracji dla poszczególnych implementacji Executora** | | | | |
| **liczba iteracji** | **SingleThreadExecutor** | **FixedThreadPool** | **CachedThreadPool** | **WorkStealingPool** |
| 100 | 232,522 | 287,0036 | 1711,262 | 303,9095 |
| 200 | 169,2725 | 225,5058 | 1357,645 | 199,8923 |
| 500 | 212,1141 | 148,7843 | 1391,865 | 145,6747 |
| 1000 | 252,2043 | 143,2917 | 1354,24 | 270,4929 |
| 1500 | 334,3941 | 167,3913 | 1426,907 | 263,2823 |
| 2000 | 330,9968 | 198,4603 | 1408,285 | 358,0644 |
| 3000 | 542,2744 | 158,9896 | 1369,702 | 512,7873 |

Obraz zawierający tekst, zrzut ekranu, linia, Wykres

Opis wygenerowany automatycznie

Na podstawie wykresu, możemy zauważyć wyraźne różnice w wydajności poszczególnych implementacji Executora w Javie (SingleThreadExecutor, FixedThreadPool, CachedThreadPool oraz WorkStealingPool) przy różnych liczbach iteracji.

SingleThreadExecutor wykazuje stabilny, lecz powolny wzrost czasu wykonania wraz z liczbą iteracji. Jest to oczekiwane, ponieważ obsługuje zadania sekwencyjnie, bez możliwości równoległego przetwarzania.

FixedThreadPool okazuje się najbardziej wydajny, szczególnie dla większych iteracji. Widać, że czasy wykonania pozostają stosunkowo niskie i stabilne, co wynika z równoległego przetwarzania z określoną liczbą wątków.

CachedThreadPool działa bardzo nieefektywnie, osiągając najdłuższy czas wykonania spośród wszystkich implementacji. Jest to spowodowane częstym tworzeniem i niszczeniem wątków, co generuje duże narzuty przy większych obciążeniach.

WorkStealingPool radzi sobie przyzwoicie, szczególnie dla mniejszych liczby iteracji, gdzie czas wykonania jest porównywalny z FixedThreadPool. Jednak dla większych iteracji jego wydajność spada, co sugeruje pewne problemy z obciążeniem wątków.

**Wnioski:**

FixedThreadPool okazuje się najlepszym wyborem, zapewniając najkrótsze i stabilne czasy wykonania dla dużych obciążeń. CachedThreadPool nie jest optymalny dla tego zadania, szczególnie przy dużej liczbie iteracji. WorkStealingPool jest wydajny przy mniejszym obciążeniu, ale nie dorównuje stabilności FixedThreadPool.

W tym przypadku , najlepiej wybrać FixedThreadPool, który efektywnie wykorzystuje zasoby wątków.

***3.Bibliografia***

* <https://www.dz5.pl/ti/java/java_skladnia.pdf> ~Jacek Rumiński, Język Java – rozdział o wątkach
* <http://brinch-hansen.net/papers/1999b.pdf> ~ Per Brinch Hansen
* <https://www.artima.com/insidejvm/ed2/threadsynch.html> ~Bill Venners, Inside the Java Virtual Machine - Charter 20 Thread Synchronization
* <https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html>
* <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html>
* <https://home.agh.edu.pl/~zobmat/2021/marchewczyk_michal/mandelbrot.html>