|  |  |
| --- | --- |
| Laboratorium nr. 6, laboratorium nr. 7 | |
| Imię i nazwisko: Miłosz Dębowski | Kierunek: Informatyka Techniczna |
| Numer albumu: 415045 | Gr. Lab.: 8 |

**Laboratorium nr. 6**

1. Cel:

* Opanowanie podstaw tworzenia wątków w Javie.
* Opanowanie podstawowych metod synchronizacji w Javie.

1. Wykonanie i wyniki

* Pobranie ze strony przedmiotu pliki z klasami: ‘Obraz.java’ oraz ‘Histogram\_test.java’.
* Analiza klasy Obraz
* Uruchomienie i testy kodu sekwencyjnego
* Rozszerzenie kodu o obliczenia równoległe - Implementacja Wariantu 1:

1. Stworzenie kodu w klasie obraz umożliwiającego obliczenie histogramu równoległego

public void calculate\_histogram\_parallel(char symbol) {  
 for(int i = 0; i < size\_n; i++) {  
 for(int j = 0; j < size\_m; j++) {  
 if(tab[i][j] == symbol) {  
 hist\_parallel[symbol - 33]++;  
 }  
 }  
 }  
}

1. Stworzenie klasy dziedziczącej po `Thread` dla obsługi wątków.

class WatekSymbol extends Thread {  
 private final Obraz obraz;  
 private final char symbol;  
  
 public WatekSymbol(Obraz obraz, char symbol) {  
 this.obraz = obraz;  
 this.symbol = symbol;  
 }  
  
 @Override  
 public void run() {  
 obraz.calculate\_histogram\_parallel(symbol);  
 obraz.print\_histogram\_parallel(symbol);  
 }  
}

1. Implementacja wątków:

* Każdy wątek przetwarza wystąpienia tylko jednego znaku w tablicy.
* Wątek korzysta z metod klasy ‘Obraz’.

1. Zarządzanie wątkami w ‘main’:

* Utworzenie wątków dla każdego znaku obecnego w tablicy.
* Uruchomienie wątków i oczekiwanie na zakończenie ich pracy.

System.*out*.println("Uruchamianie wersji równoległej");  
  
WatekSymbol[] watekArray = new WatekSymbol[94];  
for(int i = 0; i < 94; i++) {  
 watekArray[i] = new WatekSymbol(obraz\_1, (char)(i + 33));  
 watekArray[i].start();  
}  
  
for (WatekSymbol watek : watekArray) {  
 try {  
 watek.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
}

1. Dodanie funkcji w klasie ‘Obraz’, która pozwala każdemu wątkowi na wydrukowanie jego fragmentu histogramu w postaci:

```

Wątek 1: & ===============

Wątek 2: % ======

```

public synchronized void print\_histogram\_parallel(char symbol) {  
 int count = hist\_parallel[symbol - 33];  
 if (count != 0) {  
 System.*out*.print("Wątek " + symbol + ": " + symbol + " ");  
 for (int i = 0; i < count; i++) {  
 System.*out*.print("=");  
 }  
 System.*out*.println();  
 }  
}

1. Stworzenie metody w klasie ‘Obraz’, która porównuje histogramy równoległy i sekwencyjny

public void compare\_histograms() {  
 for(int i = 0; i < 94; i++) {  
 if(histogram[i] != hist\_parallel[i]) {  
 System.*out*.println("Niepoprawność dla symbolu: " + tab\_symb[i]);  
 return;  
 }  
 }  
 System.*out*.println("Histogramy są zgodne.");  
}

* Testy poprawności Wariantu 1:

1. Uruchomienie kodu.
2. Zweryfikowanie, czy dane z ‘hist\_parallel’ są zgodne z danymi z `histogram`.
3. Zweryfikowanie wydruku graficznego histogramu.

Obraz zawierający tekst, zrzut ekranu, czarne

Opis wygenerowany automatycznieObraz zawierający zrzut ekranu, tekst, czarne, Czcionka

Opis wygenerowany automatycznie

Obraz zawierający tekst, zrzut ekranu, Czcionka, design

Opis wygenerowany automatycznie

* Implementacja Wariantu 2 (dekompozycja blokowa):

1. Skopiowanie plików do nowego katalogu.
2. Zaimplementowanie wątków na podstawie `Runnable`:

* Każdy wątek przetwarza blok znaków ASCII.
* Każdy wątek aktualizuje odpowiednie fragmenty histogramu.

class HistogramWorker implements Runnable {  
 private Obraz obraz;  
 private int startSymbol;  
 private int endSymbol;  
  
 public HistogramWorker(Obraz obraz, int startSymbol, int endSymbol) {  
 this.obraz = obraz;  
 this.startSymbol = startSymbol;  
 this.endSymbol = endSymbol;  
 }  
  
 @Override  
 public void run() {  
 obraz.calculate\_histogram\_parallel\_block(startSymbol, endSymbol);  
 }  
}

public void calculate\_histogram\_parallel\_block(int startSymbol, int endSymbol) {  
 for (int i = 0; i < size\_n; i++) {  
 for (int j = 0; j < size\_m; j++) {  
 for (int k = startSymbol; k < endSymbol; k++) {  
 if (tab[i][j] == tab\_symb[k]) {  
 hist\_parallel[k]++;  
 }  
 }  
 }  
 }  
}

1. Zarządzanie wątkami w `main`:

* Utwórz wątków, przypisując im zakres znaków ASCII.
* Uruchomienie wątków i oczekiwanie na zakończenie ich pracy.

System.*out*.println("\nSet number of threads");  
int num\_threads = scanner.nextInt();  
Thread[] threads = new Thread[num\_threads];  
  
int symbolsPerThread = 94 / num\_threads;  
for (int i = 0; i < num\_threads; i++) {  
 int startSymbol = i \* symbolsPerThread;  
 int endSymbol = (i == num\_threads - 1) ? 94 : startSymbol + symbolsPerThread;  
 threads[i] = new Thread(new HistogramWorker(obraz\_1, startSymbol, endSymbol));  
 threads[i].start();  
}  
  
for (int i = 0; i < num\_threads; i++) {  
 try {  
 threads[i].join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
}  
  
System.*out*.println("\nParallel Histogram:");  
obraz\_1.print\_histogram\_parallel();

* Funkcja drukująca dla Wariantu 2:

1. Dodanie metody w klasie `Obraz`, która pozwala wątkom na wydruk ich fragmentów histogramu w postaci:

```

Wątek 1: & ===============

Wątek 2: % ======

Wątek 3: @ =========

```

public void print\_histogram\_parallel() {  
 for (int i = 0; i < 94; i++) {  
 if (hist\_parallel[i] != 0) {  
 System.*out*.print(tab\_symb[i] + " ");  
 for (int j = 0; j < hist\_parallel[i]; j++) {  
 System.*out*.print("=");  
 }  
 System.*out*.print("\n");  
 }  
 }  
}

* Testy poprawności Wariantu 2:

1. Uruchomienie kodu dla różnych zakresów ASCII.
2. Porównanie tablicy `hist\_parallel` z `histogram` w celu sprawdzenia poprawności wyników.
3. Zweryfikowanie czy wydruki wątków są poprawne i nie zakłócają się nawzajem.

Obraz zawierający zrzut ekranu, tekst, czarne

Opis wygenerowany automatycznieObraz zawierający zrzut ekranu, tekst, czarne

Opis wygenerowany automatycznie

Obraz zawierający tekst, zrzut ekranu

Opis wygenerowany automatycznie

1. Wnioski

* Działanie programu sekwencyjnego:

1. Kod sekwencyjny poprawnie oblicza histogram dla małych i dużych tablic.
2. Liczba wystąpień każdego znaku w tablicy ‘tab’ zgadza się z wartościami w tablicy ‘histogram’.
3. Działanie sekwencyjne jest proste do zaimplementowania, ale nieefektywne przy dużych tablicach.

* Działanie programu równoległego:

1. Wariant równoległy znacząco przyspiesza obliczanie histogramu dla dużych tablic, szczególnie przy większej liczbie znaków ASCII.
2. Poprawność wyników została zweryfikowana przez porównanie tablic ‘histogram’ (sekwencyjnego) i `hist\_parallel` (równoległego). Obie tablice zwracają identyczne wyniki.

* Synchronizacja wątków:

1. Synchronizacja dostępu do danych była konieczna, aby uniknąć konfliktów podczas aktualizacji tablicy ‘hist\_parallel’.
2. Synchronizowane metody klasy ‘Obraz’ pozwoliły na prawidłowe obliczanie fragmentów histogramu przez wiele wątków jednocześnie.

* Porównanie wariantów równoległych:

1. Wariant 1 (każdy wątek odpowiada za jeden znak):

* Łatwiejszy w implementacji, ponieważ każdy wątek obsługuje jedną literę.
* Wydajność może być ograniczona przy dużej liczbie znaków ASCII (liczba wątków równa liczbie znaków).
* Efektywny dla tablic zawierających mało różnorodne znaki.

1. Wariant 2 (dekompozycja blokowa):

* Bardziej elastyczny – mniejsza liczba wątków, lepsze wykorzystanie zasobów systemowych.
* Równomierny podział obciążenia między wątki.
* Wymaga bardziej złożonego zarządzania zakresem znaków w poszczególnych wątkach.
* Wydajność:

1. Kod równoległy działa szybciej dla tablic o dużych rozmiarach (np. 1000x1000).
2. Przy małych tablicach (np. 10x10) narzut związany z tworzeniem wątków sprawia, że kod równoległy jest wolniejszy od sekwencyjnego.

**Laboratorium nr. 7**

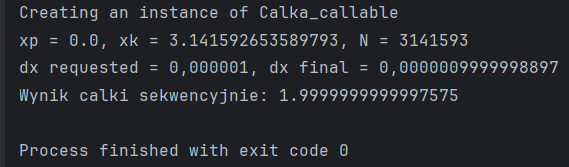
1. Cel:

* nabycie umiejętności pisania programów w języku Java z wykorzystaniem puli wątków

1. Wykonanie i wyniki

* Utworzenie katalogu roboczego
* Napisanie sekwencyjnego programu do obliczania całki metodą trapezów z użyciem klasy ‘Calka\_callable’

public class LiczenieCalkiSekwencyjne {  
 public static void main(String[] args) {  
 double start = 0;  
 double end= Math.*PI*;  
 double dx = 0.000001;  
 Calka\_callable calka = new Calka\_callable(start, end, dx);  
 System.*out*.println("Wynik calki sekwencyjnie: " + calka.compute\_integral());  
 }  
}

* Uruchomienie programu
* Pobranie i uruchomienie paczki testowej
* Modyfikacja programu na wersję równoległą z wykorzystaniem ExecutorService:

1. Podzielenie przedziału całkowania (0, Math.PI) na tyle podprzedziałów, ile jest zadań. Szerokość podprzedziałów jest niezależna od parametru ‘dx’.
2. Tworzenie zadań i modyfikacja klasy ‘Calka\_callable’

* Utworzenie obiekt klasy ‘Calka\_callable’ dla każdego podprzedziału i przekazanie go do puli wątków.
* Zmodyfikowanie klasy ‘Calka\_callable’:
* Odkomentowanie nagłówka umożliwiającego wykorzystanie obiektu jako zadania w puli wątków.
* Uzupełnienie kodu klasy o wymaganą przez interfejs funkcję ‘call()’.

1. Niezależność parametrów

* Liczba wątków i liczba zadań są niezależne.
* Liczba wątków powinna być związana z liczbą rdzeni procesora.
* Liczba zadań powinna być kilka razy większa od liczby wątków, aby równoważyć obciążenie.

1. Tworzenie i wykonywanie zadań

* Zadania są tworzone i przekazywane do wykonania w jednej pętli.

1. Odbieranie wyników

* Korzystając z interfejsu ’Future’, wyniki są odbierane w oddzielnej pętli, aby umożliwić działanie równoległe.
* Kod programu

1. Klasa ‘Liczeniecalki’

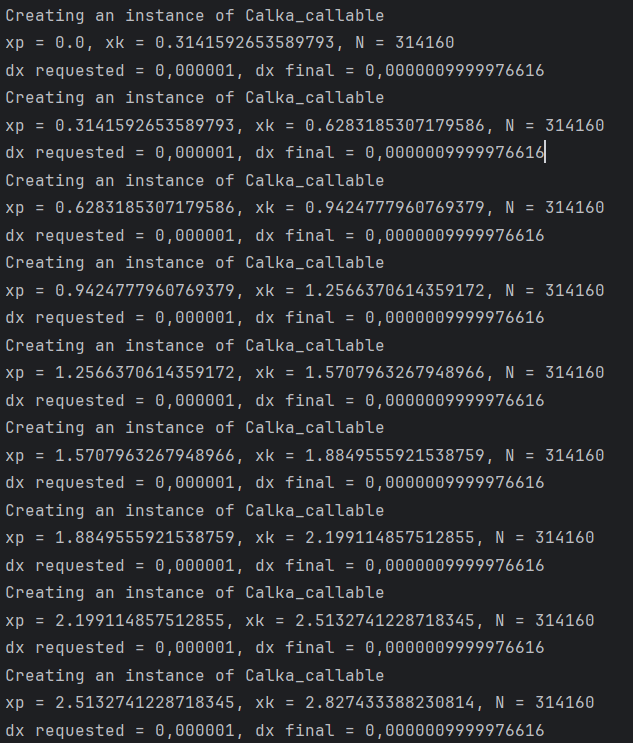
import java.util.concurrent.\*;  
import java.util.List;  
import java.util.ArrayList;  
  
public class Liczeniecalki {  
  
 private static final int *NTHREADS* = 10;   
 public static void main(String[] args) {  
 ExecutorService executor = Executors.*newFixedThreadPool*(*NTHREADS*); //tworzenie wykonawcy o zadanej puli wątków  
 List<Future<Double>> listaWynikow = new ArrayList<Future<Double>>(); // lista w ktorej zapisujemy obiekty future  
  
 //Przedział  
 double start = 0;  
 double end = Math.*PI*;  
  
 double task = 50; //ilość zadań  
 double dx = 0.000001; //dokładność  
  
 double przedzialPerZadanie = (end - start) / task; //przedział dla każdego pojedynczego zadania  
  
 double kX = przedzialPerZadanie; //zmienna pomocnicza  
  
 for (double pX = start; pX < end; pX += przedzialPerZadanie){  
  
 if(kX > end){kX = end;} // w przypadku bledu zaokraglen ten if to wychwyci  
  
 Callable<Double> calkacallable = new Calka\_callable(pX, kX, dx);  
 Future<Double> future = executor.submit(calkacallable);  
 listaWynikow.add(future);   
 kX += przedzialPerZadanie; //przejscie na inny przedział dla kolejnego wątku z puli  
 }  
  
 double wynik = 0;  
  
 for (Future<Double> future : listaWynikow){  
 try {  
 wynik += future.get(); // uzyskanie wyników metody call  
 }  
 catch (InterruptedException | ExecutionException e){  
 e.printStackTrace();  
 }  
   
 }  
 executor.shutdown();  
  
 // Wait until all threads finish  
 while (!executor.isTerminated()) {}  
 System.*out*.println("Finished all threads");  
 System.*out*.println("Wynik calkowania rownolegle: " + wynik);  
   
 }  
 }

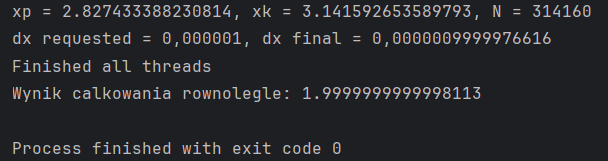
1. Kod dodany do klasy ‘Calka\_callable’

public class Calka\_callable implements Callable<Double>{

public Double call() throws Exception {   
 return compute\_integral();  
}

* Uruchomienie i sprawdzenie wyników z metodą sekwencyjną

1. Uruchomienie zmodyfikowanego programu dla 10 zadań.



1. Wnioski

* Poprawność obliczeń

1. Wersja sekwencyjna programu dostarczyła wyniki zgodne z oczekiwaniami, co zostało zweryfikowane przez porównanie z dokładnym rozwiązaniem analitycznym dla funkcji testowej `sin(x)` w przedziale (0, pi).
2. Wersja równoległa programu również zwróciła poprawne wyniki, zgodne z wersją sekwencyjną, co świadczy o prawidłowym podziale pracy na podprzedziały i ich sumowaniu.

* Wydajność

1. Zastosowanie puli wątków znacząco przyspieszyło obliczenia dla dużej liczby zadań w porównaniu z wersją sekwencyjną, szczególnie na maszynach z wielordzeniowymi procesorami.
2. Wydajność skalowała się wraz ze wzrostem liczby wątków, ale przy nadmiernie dużej liczbie wątków względem dostępnych rdzeni zauważono efekt zmniejszenia efektywności.

* Balans obciążenia

- Dzięki podziałowi przedziału całkowania na wiele zadań obciążenie procesora zostało dobrze zbalansowane.

- Nawet jeśli niektóre zadania wymagały minimalnie większej liczby operacji, mechanizm kolejkowania zadań w ‘ExecutorService’ umożliwił efektywne wykorzystanie zasobów.