Sprawozdanie – Laboratorium 7

Temat: Programowanie równoległe w Javie (pula wątków)

1. Cel laboratorium

Celem zajęć było nabycie umiejętności wykorzystania puli wątków w języku Java w celu równoległego obliczania całki oznaczonej metodą trapezów.

2. Wersja sekwencyjna

Zaimplementowano klasę Calka_callable, która:
- przyjmuje jako parametry przedział całkowania oraz dokładność dx,
- wewnętrznie oblicza liczbę trapezów,
- zawiera metodę compute_integral() obliczającą wartość całki w zadanym przedziale.

W wersji sekwencyjnej obliczanie odbywało się w jednym wątku, na całym przedziale funkcji.

3. Wersja równoległa z pulą wątków

Z wykorzystaniem ExecutorService i Executors.newFixedThreadPool() utworzono pulę o zadanej liczbie wątków. Przedział całkowania dzielony jest na niezależne podprzedziały, dla których tworzono osobne zadania obliczające całkę.

- Każde zadanie było reprezentowane przez zmodyfikowaną klasę Calka_callable, implementującą interfejs Callable<Double>.
- Zadania przekazywano do puli w jednej pętli, a wyniki zbierano w drugiej pętli z użyciem interfejsu Future.

4. Parametry programu

Program przyjmował trzy niezależne parametry:
- dx – dokładność całkowania (wysokość trapezów),
- liczba wątków – ustalana zależnie od liczby rdzeni procesora,
- liczba zadań – typowo większa od liczby wątków, by umożliwić efektywne równoważenie obciążenia.

5. Testy i poprawność

Program przetestowano na przykładzie funkcji sin(x) w przedziale (0, π), gdzie znany wynik to 2.

```
→  java_executor_test 2 git:(master) ✗ echo "Mikita Shmialiou"
Mikita Shmialiou
→  java_executor_test 2 git:(master) ✗ java IntegralExecutor

∫_[0.0, 3.14] sin(x) dx ≈ 1.999999998334 (wartość dokładna: 2.0)
→  java_executor_test 2 git:(master) ✗
```

6. Wnioski

- Użycie ExecutorService pozwala na łatwe i efektywne zarządzanie pulą wątków.
- Dzięki dzieleniu obliczeń na mniejsze zadania, możliwe jest wykorzystanie wielu rdzeni procesora, co znacząco przyspiesza obliczenia przy większych przedziałach i dokładności.
- Oddzielenie liczby zadań od liczby wątków umożliwia elastyczne i wydajne zarządzanie obciążeniem.

Kod: IntegralExecutor.java

```
import java.util.concurrent. *;
import java.util. *;

public class IntegralExecutor {

   // --- Niezależne parametry ---
   private static final double DX       = 1.0e-4;                    // dokładność
   private static final int    NTHREADS = Runtime.getRuntime().availableProcessors();
   private static final int    NTASKS   = NTHREADS * 4;                  // np. 4× więcej zadań

   public static void main(String[] args) throws InterruptedException, ExecutionException {

      double XP = 0.0;                  // początek przedziału
      double XK = Math.PI;                 // koniec przedziału

      ExecutorService pool = Executors.newFixedThreadPool(NTHREADS);
      List<Future<Double>> futures = new ArrayList<>(NTASKS);

      // --- Pętla 1: tworzenie i submit zadań ---
      double subWidth = (XK - XP) / NTASKS;
      for (int i = 0; i < NTASKS; i++) {
         double subXp = XP + i * subWidth;
         double subXk = (i == NTASKS - 1) ? XK : subXp + subWidth; // ostatni domyka przedział

         Calka_callable task = new Calka_callable(subXp, subXk, DX);
         futures.add(pool.submit(task));
      }

      // --- Pętla 2: odbiór wyników ---
      double total = 0.0;
      for (Future<Double> f : futures) {
         total += f.get(); // blokuje do ukończenia zadania
      }
      pool.shutdown();
      // --- Raport ---
      System.out.printf(
```

```
            "∫_[%.1f, %.2f] sin(x) dx ≈ %.12f (wartość dokładna: 2.0)%n",
            XP, XK, total
        );
    }
}
```