

# Laboratorium 7

Imię i nazwisko  
Łukasz Sawina

## Cel

Celem ćwiczenia było poznanie pisania programów w języku Java z wykorzystaniem puli wątków.

## Obliczanie całki przy pomocy puli wątków

W pierwszej części naszym zadaniem było napisanie programu liczącego całkę sekwencyjnie wykorzystując metodę trapezów. W konstruktorze klasy do liczenia całki przyjmuje ona takie parametry jak początek, koniec przedziału oraz wartość  $dx$ . Następnie w metodzie `compute_integral()` obliczana jest nasza całka.

```
public double compute_integral() {  
    double calka = 0;  
    int i;  
    for(i=0; i<N; i++){  
        double x1 = xp+i*dx;  
        double x2 = x1+dx;  
        calka += ((getFunction(x1) + getFunction(x2))/2.)*dx;  
    }  
    System.out.println("Calka czastkowa: " +
```

W kodzie wyświetlany jest napis Calka czastkowa..., jest on wykorzystywany w dalszej części przy zrównoleglaniu zadania.

Wywołanie funkcji wygląda następująco:

```
static double xs = 0;  
static double xe = Math.PI;  
static double dx = 0.00001;  
  
public static void main(String[] args)  
{  
    Calka_callable calka = new Calka_callable(xs, xe, dx);  
    System.out.println("Wynik calki: " + calka.compute_int
```

Wynikiem programu przy liczeniu całki z funkcji  $\sin(x)$  w przedziale  $[0, \pi]$  jest:

```
xp = 0.0, xk = 3.141592653589793, N = 3141593
dx requested = 1.0E-6, dx final = 9.99998897342186E-7
Całka czastkowa: 1.9999999999997575
Wynik całki: 1.9999999999997575
```

Następnie naszą klasę musieliśmy zmienić, aby implementowała interfejs *Callable*, co wymusza na niej posiadanie metody *call()*, która w moim programie wywołuje metodę *compute\_integral()*. Nie wymaga ona żadnych parametrów, ponieważ wszystkie wartości są przekazywane do konstruktora.

Tym razem do obliczenia całki wykorzystamy pulę wątków, do której będą przekazywane różne zadania (w tym przypadku różne przedziały całki) i jego wyniki będą zapisywane w liście, a na końcu sumowane.

```
static double xs = 0;
static double xe = Math.PI;
static double dx = 0.000001;
private static final int NTHREADS = 4;
private static final int NTASK = 40;

public static void main(String[] args)
{
    double offset = (xe/NTASK);

    ExecutorService executor = Executors.newFixedThreadPool(NTHREADS);
    List<Future<Double>> wyniki_czastkowe = new ArrayList<Future<Double>>();

    for (int i = 0; i < NTASK; i++) {
        Callable<Double> calka = new Calka_callable(i*offset, (i+1)*offset, dx);
        Future<Double> wynik = executor.submit(calka);
        wyniki_czastkowe.add(wynik);
    }

    double wynik_Calki = 0;

    for(Future<Double> wynik : wyniki_czastkowe){
        try {
            wynik_Calki+=wynik.get();
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
    executor.shutdown();

    while(!executor.isTerminat
```

W programie wykorzystujemy 4 rdzenie oraz cały obszar dzielimy na 40 podzadań. Na początku obliczany jest *offset* dla zadań, czyli jaki obszar dostanie każdy z nich.

Następnie tworzony jest *executor*, który będzie wykorzystywał nasze 4 wątki do zarządzania pulą zadań oraz Lista wyników cząstkowych. Jak można zauważyć lista jest typu *Future<Double>*, ponieważ wyniki z puli wątków właśnie w takim typie są zapisywane.

W kolejnej części w pętli tworzone są obiekty naszej klasy liczące całkę i przesyłane są w konstruktorze początek, koniec zakresu całkowania dla zadania oraz wartość  $dx$ . W zmiennej wynik zapisywany jest wynik dla danego podzadania przez wywołanie na *executor* metody *submit(calka)*. Metoda *submit* daje nam opcję zwrócenia wyniku z naszego zadania. Następnie wynik jest dodawany do listy wyników cząstkowych.

Na końcu w pętli *foreach* przechodzimy po wszystkich wynikach z zadań i próbujemy dostać ich wynik do jednej zmiennej. Zlecamy naszemu *executorowi* zakończenie pracy przy pomocy metody *shutdown()* i oczekujemy na zakończenie (w celu upewnienia się, że wszystkie zdania zostały wykonane).

Wynikiem naszego programu jest:

```
Całka czastkowa: 0.04362316997914111
Całka czastkowa: 0.038366359834272
Całka czastkowa: 0.03287300832291645
Całka czastkowa: 0.02717698378386477
Całka czastkowa: 0.021313404102521614
Całka czastkowa: 0.015318420197459848
Całka czastkowa: 0.003082666266871751
Całka czastkowa: 0.00922899313798956
Wynik całki: 1.9999999999998364
```

Jak widać dodatkowo dostajemy informację o cząstkowych wartościach całki, rzecz możliwa do pominięcia tak naprawdę.

## Sortowanie przy pomocy ForkJoinPool

Kolejnym programem do napisania było sortowanie przy pomocy scalania. Program działa na zasadzie Dziel i zwyciężaj, czyli tablicę dzielimy na coraz to mniejsze tablice, aż dostaniemy tablicę jednoelementową. W efekcie tablica jednoelementowa jest już tablicą posortowaną. Następnie scalamy dwie posortowane tablice w większą i tak aż wrócimy do początkowego rozmiaru.

Zdanie scalania wykonywane jest w klasie *DivideTask*, która dziedziczy po *RecursiveTask<>*. Metoda do scalania dwóch tablic została już napisana, dlatego jej nie będę analizował.

Głównym zadaniem było napisanie metody *compute()*, której zadaniem jest podział tablicy na dwa oraz wykonanie rekurencji na samej sobie, a ostatecznie scalenie wyników.

Jedną z najważniejszych części metody *compute* jest warunek zakończenia, w tym przypadku podział na mniejsze jest kończony, gdy nasza tablica będzie miała rozmiar 1.

```

protected int[] compute() {
    if (arrayToDivide.length > 1)
    {
        int mid_index = (arrayToDivide.length) / 2;

        int[] t1 = new int[mid_index];
        int[] t2 = new int[arrayToDivide.length - mid_index];

        for (int i = 0; i < mid_index; i++) {
            t1[i] = arrayToDivide[i];
        }

        for (int i = mid_index; i < arrayToDivide.length; i++) {
            t2[i - mid_index] = arrayToDivide[i];
        }

        DivideTask task1 = new DivideTask(t1, 0, mid_index);
        DivideTask task2 = new DivideTask(t2, mid_index, t2.length);

        task1.fork();
        task2.fork();

        int[] tab1 = task1.join();
        int[] tab2 = task2.join();

        scal_tab(tab1, tab2, arrayToDivide);
    }
    return arrayToDivide;
}

```

W funkcji kolejno obliczany jest indeks środkowego elementu naszej tablicy, później wykorzystując ten indeks tworzymy dwie tablice i przepisujemy do nich odpowiednie elementy (zwykły podział na dwie tablice).

Kolejno tworzone są dwa nowe zadania do których przekazywane są podzielone tablice oraz ich indeksy początku oraz końca.

Funkcja jest wykonywana rekurencyjnie w linijce *task1.fork()* oraz *task2.fork()*. Wewnątrz każdego z zadań wykonywane są ponownie powyższe kroki.

Później wynikowe tablice z zdania są zwracane do *tab1* oraz *tab2* i scalane przy pomocy dostarczonej funkcji scalania dwóch tablic.

Uruchomienie programu wygląda następująco:

```

public static void main(String[] args) {

    int[] numbers = new int[] {2, 1, 3, 4, 5, -2, 321321};

    DivideTask task = new DivideTask(numbers, 0, numbers.length);
    ForkJoinPool forkJoinPool = new ForkJoinPool();

    forkJoinPool.execute(task);
    int[] wynik = task.join();

    for(int number : wynik)
        System.out

```

Tworzona jest tablica, w tym przypadku jest utworzona statycznie, ale można również utworzyć ją z losowych liczb.

Tworzony jest obiekt naszej klasy do sortowania oraz obiekt klasy *ForkJoinPool()*. Przy pomocy metody *execute(task)* uruchamiamy metodę *compute()* w naszej klasie do sortowania i przy pomocy *join()* oczekujemy na zwrócenie wyniku.

Następnie nasza tablica jest wyświetlana.

```

Nieposortowana tablica
2 1 3 4 5 -2 321321
Posortowana tablica
-2 1 2 3 4 5 321321

```

Tablica jak widać została poprawnie posortowana.

## Obliczanie całki przy pomocy puli wątków z wykorzystaniem interfejsu Runnable

W tym zadaniu ponownie obliczamy całkę, jednak klasa licząca całkę nie implementuje interfejsu *Callable* a *Runnable*. Pojawia się tutaj przez to trudność, ponieważ *Runnable* nie zwraca wyniku, dlatego potrzebujemy innego mechanizmu do pobrania wynikowych całek cząstkowych.

Do tego utworzyłem klasę *PrzekazywaczDanych*, która implementuje nowy interfejs *IprzekazywaczDanych*.

```

public class PrzekazywaczDanych implements IPrzekazywaczDanych{
    private double wynikCalki = 0;
    @Override
    public synchronized void ZapiszWynikCzastkowy(double wynik) {
        wynikCalki += wynik;
    }

    public double ZwrocWynik() {
        re

```

W programie mamy dwie metody, jedną synchronizowaną *ZapiszWynikCzastkowy(double wynik)*, która służy do bezpiecznego zapisania wyniku cząstkowego oraz *ZwrocWynik()*, która zwraca wynik całki.

```

public void compute_integral() {
    double calka = 0;
    int i;
    for(i=0; i<N; i++){
        double x1 = xp+i*dx;
        double x2 = x1+dx;
        calka += ((getFunction(x1) + getFunction(x2))/2.)*dx;
    }
    System.out.println("Calka czastkowa: " + calka);

    przekazywaczDanych

```

Metoda licząca całkę działa tak samo jak w pierwszym programie, tylko wynik jest przekazywany do obiektu *przekazywaczDanych*.

Dodatkowymi różnicami jest to, że klasa do liczenia całki musi posiadać metodę *run()* zamiast *call()* oraz w konstruktorze dodatkowo przekazywany jest obiekt *PrzekazywaczDanych*.

Wykonanie całego programu również dużo się nie różni od pierwszego programu.

```

PrzekazywaczDanych przekazywacz = new PrzekazywaczDanych();
ExecutorService executor = Executors.newFixedThreadPool(NTHREADS);
List<Future> wyniki_czastkowe = new ArrayList<Future>();
double offset = (xe/NTASK);

for (int i = 0; i < NTASK; i++) {
    Runnable calka = new Calka_runnable((double) i * offset, (double) (i + 1) *
offset, dx, przekazywacz);
    Future future = executor.submit(calka);
    wyniki_czastkowe.add(future);
}

Iterator<Future> iterator = wyniki_czastkowe.iterator();

while(iterator.hasNext()){
    Future future = iterator.next();
    try {
        future.get();
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}

executor.shutdown();
while(!executor.isTerminated()){ }
System.out.println("Wynik calki: " + przekazywacz.ZwrocWynik() );

```

Na początku mamy tworzony obiekt `PrzekazywaczDanych`, executor z przekazaną ilością rdzeni oraz lista na wyniki cząstkowe. W tym przypadku Lista wyników jest typu `Future`, ponieważ nie zwraca ona żadnego wyniku.

Następnie w pętli tworzone są wszystkie zadania, są one submitowane, oraz ich wyniki dodawane do listy wniosków.

Tworzony jest dodatkowo iterator po wynikach cząstkowych, dzięki któremu możemy przejść po wszystkich zadaniach i wywołać na nich metodę `get()`.

Na końcu executor zleca zakończenie zadań, oczekuje na zakończenie oraz z `PrzekazywaczaDanych` pobierany jest wynik całki.

Wynikiem programu jest:

```
Całka cząstkowa: 0.04362316997914111
Całka cząstkowa: 0.038366359834272
Całka cząstkowa: 0.03287300832291645
Całka cząstkowa: 0.02717698378386477
Całka cząstkowa: 0.021313404102521614
Całka cząstkowa: 0.015318420197459848
Całka cząstkowa: 0.00922899313798956
Całka cząstkowa: 0.003082666266871751
Wynik całki: 1.999999999998364
```

## Program histogramu z wykorzystaniem puli wątków

W tym zadaniu musimy wykorzystać program z poprzednich laboratoriów do liczenia histogramu z obrazu i przekształcić go tak aby wykorzystywał pulę wątków. W tym przypadku wykorzystuję wersję do podziału obrazu 2D.

Zmian w samej klasie do liczenia histogramu oraz klasy `Obrazu` nie ma, jedyna zmiana pojawia się w głównym programie. Część pobierania wymiarów obrazu oraz jego generowanie jest identyczne.

```
int num_threads = scanner.nextInt();
```

```
ExecutorService executor = Executors.newFixedThreadPool(num_threads);
```

```
int w_offset = (int)ceil(obraz_1.getSizeN()/((double)num_threads/2));
```

```
int k_offset = (int)ceil(obraz_1.getSizeM()/((double)num_threads/2));
```

```
for (int i = 0, k = 0; i < num_threads/2; i++) {  
    for(int j = 0; j < num_threads/2; j++, k++)  
    {
```

```
        int w_start = i*w_offset;
```

```
        int w_end = (i+1)*w_offset;
```

```
        int k_start = j*k_offset;
```

```
        int k_end = (j+1)*k_offset;
```

```
        if(w_end > obraz_1.getSizeN())  
            w_end = obraz_1.getSizeN();
```

```
        if(k_end > obraz_1.getSizeM())  
            k_end = obraz_1.getSizeM();
```

```
        WatekRunnable2D watekRunnable = new WatekRunnable2D(obraz_1,  
w_start, w_end, k_start, k_end);  
        executor.execute(watekRunnable);
```

```
    }  
}
```

```
executor.shutdown();
```

```
while(!executor.isTerminated()) {}
```

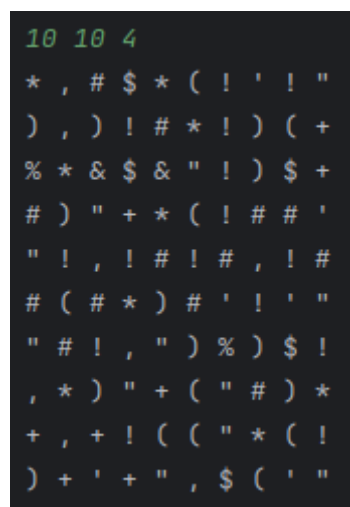
Ponownie na początku tworzony jest obiekt executor z przekazaną ilością wątków. Tym razem nie potrzebujemy tworzyć tablicy dla podzadań, tylko tworzony jest obiekt dla każdego z podobrazu z odpowiednimi przedziałami i wykonywany jest za pomocą *execute()*.

Na końcu executor kończy pracę oraz oczekujemy na wykonanie wszystkich zadań.

W dalszej części programu histogram jest wyświetlany oraz sprawdzana jest jego poprawność.

Wynikiem programu jest np.:

Obraz, dla którego tworzony jest histogram:



```
10 10 4  
* , # $ * ( ! ' ! "  
) , ) ! # * ! ) ( +  
% * & $ & " ! ) $ +  
# ) " + * ( ! # # '  
" ! , ! # ! # , ! #  
# ( # * ) # ' ! ' "  
" # ! , " ) % ) $ !  
, * ) " + ( " # ) *  
+ , + ! ( ( " * ( !  
) + ' + " , $ ( ' "
```



Histogram obrazu:

```
Znak: ! =====
Znak: " =====
Znak: # =====
Znak: $ =====
Znak: % ==
Znak: & ==
Znak: ' =====
Znak: ( =====
Znak: ) =====
Znak: * =====
Znak: + =====
Znak: , =====
Oba histogramy są identyczne
```

## Wnioski

W tych labolatoriach wykorzystaliśmy już istniejący mechanizm do zadań wielowątkowych, czyli pulę wątków. Dzięki temu nie musieliśmy sami zarządzać naszymi wątkami, tylko mamy od tego cały system, który sam dzieli nasze zadania na wątki. Dodatkowo takie wykorzystanie ułatwia nam synchronizowanie wyników, ponieważ możemy wyniki z danego zadania zapisać w zmiennej typu `Future<>` i np. zapisywać wszystkie w liście lub tablicy, a następnie je łączyć. Nie musimy przez to tworzyć metod *synchronized* lub zabezpieczać jakiejś zmiennej.

Dodatkowo poznaliśmy mechanizm rekurencyjnej pracy z wieloma wątkami, jak tworzyć metody rekurencyjne oraz jak je wywoływać.