JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 16 (2020) 241–246
Received: 7 June 2020
Accepted: 24 June 2020

Comparative analysis of Kotlin coroutines with Java and Scala in parallel programming

Analiza porównawcza współprogramów języka Kotlin z językami Java i Scala w przetwarzaniu równoległym

Adrian Andrzej Zieliński*

[a] *Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland*

Abstract

The article presents a comparison of Kotlin coroutines with analogous solutions in Java and Scala in parallel programming using chosen metric and non-metric criteria. For that purpose, a multi-module project with corresponding implementations of selected algorithms in all of the three languages was created and then analyzed. The studies were preceded by a description of the created project.

Keywords: kotlin; java; scala; parallel programming

Streszczenie

Artykuł prezentuje porównanie wykorzystania współprogramów języka Kotlin w przetwarzaniu równoległym do analogicznych rozwiązań w Javie i Scali względem wybranych kryteriów mierzalnych i niemierzalnych. W tym celu stworzono oraz przeanalizowano wielomodułową aplikację z odpowiadającymi sobie implementacjami wyselekcjonowanych algorytmów w trzech wspomnianych językach. Analiza poprzedzona została opisem utworzonego projektu.

Słowa kluczowe: kotlin; java; scala; programowanie równoległe

*Corresponding author

Email address: **adrian.zielinski@pollub.edu.pl** (A. A. Zieliński)

©Published under Creative Common License (CC BY-SA v4.0)

1. Wstęp

Pomimo leciwości konceptu programowania równoległego, który powstał jeszcze w XIX w., to jego realne zastosowanie w informatyce datuje się dopiero na drugą połowę wieku XX za sprawą popularyzacji procesorów wielordzeniowych [1]. Obecnie nastąpiło już w tej materii takie przesycenie, że koncept ten może być wcielany w życie na wyższych poziomach abstrakcji dzięki mnogości dostępnych platform programistycznych.

Jedną z najciekawszych cech Kotlina, która jest jednocześnie przedmiotem analizy w niniejszym artykule, jest biblioteka kotlinx.coroutines, która została włączona do stabilnego wydania języka wraz z wersją 1.3. Oferuje ona funkcjonalność współprogramów zaproponowanych jeszcze w latach 60 XX wieku przez Conwaya [2] na poziomie języka, z możliwościami równoległego wykonania, podczas gdy Java i Scala operują na klasycznych konceptach wątków czy nieco nowszych obietnic.

Jako, że Kotlin nie jest pierwszym, ani nawet najpopularniejszym językiem oferującym omawianą funkcjonalność, zasadne staje się pytanie jak jego nowe podejście do współprogramów wypada na tle technologii, które zdążyły się już zadomowić na rynku. Niniejsza praca stara się odpowiedzieć na to pytanie poprzez bezpośrednie porównanie implementacji współprogramów w Kotlinie oraz technik przetwarzania równoległego w Javie i Scali, jako, że wszystkie z wymienionych języków mogą działać pod wirtualną maszyną Javy GraalVM.

2. Przegląd literatury

Przez ostatnie kilka lat, tematyka współprogramów przeżywa swój renesans. Coraz bardziej skomplikowane programy oraz rozwój sprzętowy w kierunku przetwarzania wielordzeniowego i wielowątkowego wymusza niejako poszukiwanie coraz to wydajniejszych, bardziej rozbudowanych czy łatwiejszych w użyciu technik przetwarzania asynchronicznego i równoległego.

W 2009 roku De Moura i Ierusalimschy przekonywali, że przywrócenie współprogramów do łask po wielu latach zapomnienia odbyłoby się z korzyścią dla wszystkich. W swojej pracy przedstawili klasyfikację podejść do implementacji tego paradygmatu, argumenty przemawiające za bądź przeciw danemu podejściu oraz wprowadzili nową koncepcję współprogramów w pełni asymetrycznych, która łączy w sobie cechy poprzedników [3].

Racordon w swojej pracy porusza natomiast problem braku możliwości programowania z wykorzystaniem paradygmatu współprogramów w wielu współczesnych językach programowania i proponuje uniwersalny model implementacyjny tychże na bazie funkcji wyższego rzędu w językach je posiadających, prezentując przykłady w języku JavaScript [4].

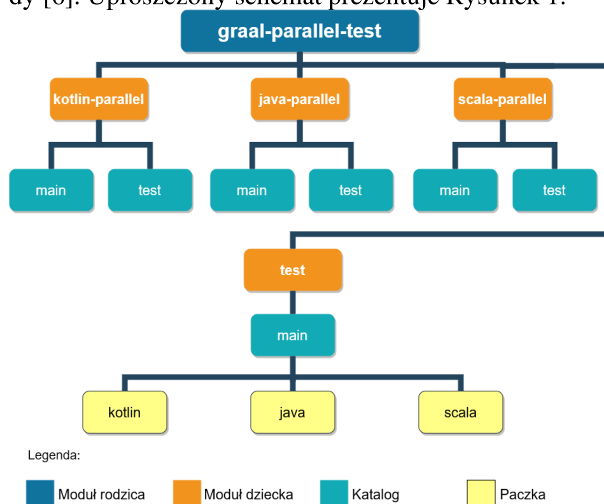
Z kolei Ohlsson i Leffler zwracają uwagę na możliwość implementacji współprogramów w Javie poprzez formę rozszerzenia językowego. Autorzy słusznie zauważają, że istnieją problemy, które naturalniej jest rozwiązać wykorzystując opisywany paradygmat i na przykładzie własnego kodu napisanego w standardowej

Javie pokazują, że przemodelowanie go na wykorzystywanie wcześniej utworzonego rozszerzenia współprogramów pozwala na uzyskanie większej zwięzłości i czytelności [5]. Przytoczone powyżej prace ukazują zwiększone zainteresowanie tematyką współprogramów na przestrzeni poprzednich lat wśród badaczy z dziedziny programowania. Jednocześnie, bezpośrednie porównanie współprogramów z klasycznymi technikami przetwarzania równoległego nie doczekało się obszerniejszej analizy w obrębie maszyny wirtualnej Javy. Niniejszy artykuł rzuca na ten temat nowe spojrzenie, mogące pomóc w wyborze odpowiedniej do danego problemu technologii.

3. Projekt testowy

3.1. Opis

Stworzona z wykorzystaniem systemu budującego i zarządzającego zależnościami Maven aplikacja podzielona została na cztery główne moduły – kotlin-parallel, java-parallel, scala-parallel oraz test. Pierwsze trzy składają się z logiki zdefiniowanej kolejno w Kotlinie, Javie oraz Scali i zawierają możliwie najbardziej zbliżone sobie implementacje trzech wybranych, zrównoleglonych algorytmów – znajdowania liczb pierwszych, wyznaczania otoczki wypukłej punktów w przestrzeni dwuwymiarowej oraz wyliczania transformaty Fouriera. Dodatkowo wytworzono zestaw testów jednostkowych sprawdzających poprawność logiczną. Ostatnia część o nazwie test zawiera odpowiednio podzieloną grupę testów wydajnościowych dla wszystkich badanych implementacji. Do celów jej zmierzenia wykorzystano bibliotekę Java Microbenchmark Harness autorstwa Oracle'a. JMH jest narzędziem przeznaczonym dla platformy Java do wykonywania powtarzalnych i miarodajnych mikro testów wydajnościowych, tj. takich, których zadaniem jest pomiar możliwie najmniejszej części systemu – najlepiej pojedynczej metody [6]. Uproszczony schemat prezentuje Rysunek 1.



Rysunek 1: Uproszczony schemat projektu

3.2. Wykorzystane algorytmy

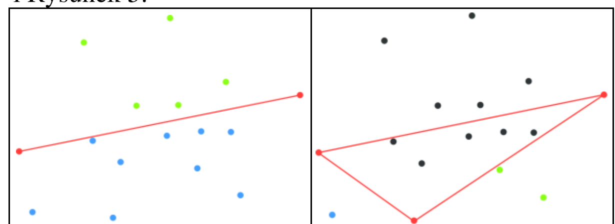
W celu oddania możliwie jak najszerszego zakresu wykorzystania obliczeń równoległych, wybrane zostały trzy algorytmy oddające różne przypadki czy scenariusze testowe. Pierwszym, a zarazem logicznie najprostszym algorytmem jest Sito Eratostenesa służące do wyznaczania liczb pierwszych w przedziale [7]. Blokowa wersja Sita idealnie nadaje się do równoległego wykonywania niemal wszystkich bloków liczb z zakresu i tym samym oddaje pierwszy zakładany przypadek użycia równoległości. Poniższy Rysunek 2. przedstawia zasadę działania dla dwóch pierwszych kroków Sita.

	2	3	4	5	6	7	8	9	10	Prime
11	12	13	14	15	16	17	18	19	20	2
21	22	23	24	25	26	27	28	29	30	
31	32	33	34	35	36	37	38	39	40	

	2	3	4	5	6	7	8	9	10	Prime
11	12	13	14	15	16	17	18	19	20	2 3
21	22	23	24	25	26	27	28	29	30	
31	32	33	34	35	36	37	38	39	40	

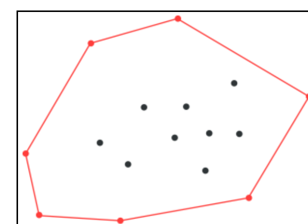
Rysunek 2: Pierwsze dwa kroki podstawowego Sita [8]

Jako drugi algorytm wybrano Quickhull służący do wyznaczania otoczki wypukłej skończonej liczby punktów w przestrzeni. Ze względu na swoją rekursywną naturę algorytm przetwarzany jest w formie drzewa, które rozwiązywane jest od liści do korzenia [9]. Dzięki temu idealnie wpasowuje się w przypadek dużej liczby małych, współbieżnych zadań, które wstrzymywane są na czas wykonywania swoich zależności i następnie wznawiane w celu scalenia wyników. Kroki działania algorytmu pokazują poniższe Rysunek 3., Rysunek 4. i Rysunek 5.



Rysunek 3: Wyznaczenie dwóch zbiorów początkowych [10]

Rysunek 4: Znalezienie punktu maksymalnego w jednym ze zbiorów [11]



Rysunek 5: Wynik rekursji w momencie braku kolejnych zewnętrznych punktów [12]

Ostatnim z wybranych przypadków testowych jest algorytm wyznaczania widma do analizy sygnału audio w nieskompresowanym formacie wav z wykorzystaniem szybkiej transformacji Fouriera (FFT). Rozpatry-

wany problem jest o tyle różny od dwóch pozostałych, że angażuje dodatkowo blokujące zadanie odczytu z pliku, przez co wszystkie zadania, mogące być teoretycznie wykonywane równolegle, są częściowo ograniczone przez możliwości dostarczania kolejnych partii przez strumień danych.

4. Kryteria oceny

4.1. Wydajność

Aspekt wydajnościowy jest po prawdzie jednym z łatwiej mierzalnych w środowisku JVM ze względu na dostępność biblioteki JMH od Oracle'a. Pozwala ona bowiem tworzyć powtarzalne testy w tak trudnym środowisku jak maszyna wirtualna stosująca optymalizacje kompilowanego na bieżąco kodu i to na wielu poziomach, zaczynając od wstępnej optymalizacji podczas pierwszego wykonania.

Wszystkie benchmarki przypadków testowych oparte zostały o wcześniej przedstawioną bibliotekę JMH i zawarte zostały w module test, który przedstawiony jest na Rysunek 1.

Każdy przypadek testowy wykonany zaś został 10 razy z jednakowymi ustawieniami adnotacji na klasach testowych.

4.2. Długość kodu

W przypadku tytułowej metryki postanowiono przyjąć pewne założenia pozwalające na dokonanie miarodajnego porównania. Ocenie podlegała całościowa długość napisanego dla danej implementacji kodu w liniach, przy czym przyjęto, że mniejsza liczba linii oznacza lepszy, tj. czytelniejszy kod. Założenie to wynika głównie z niechęci programistów do technologii wymuszających na nich pisanie tzw. boilerplate code czyli w wolnym tłumaczeniu kodu potrzebnego do rozwiązania problemu algorytmicznego w danym języku, który jednak mógłby być całkowicie zbędny w innych warunkach, np. w innym, bardziej zwięzłym języku. Zaowocowało to nawet, nie licząc mnogości technologii promujących zwięzły kod, poszukiwaniem automatyzowanych rozwiązań tego problemu [13].

Przy formatowaniu źródeł utworzonych programów postawiono na jednolite zasady. Długość linii nie większą niż 120 znaków, z tolerancją ± 10 oraz klamry dla wszystkich instrukcji ich wymagających, np. if, while, for czy definicji metod i funkcji. Pozostałe ustawienia miały domyślne wartości wykorzystanego środowiska programistycznego IntelliJ IDEA 2019.3.4.

4.3. Czas kompilacji

Ważnym czynnikiem przy doborze technologii jest również całkowity czas kompilacji projektu, którego wahania mogą wpływać na wiele kwestii począwszy od produktywności programisty a skończywszy na szybkości integracji rozwiązania np. przy continous integration.

Zmierzenie czasu kompilacji poszczególnych części badanej aplikacji stało się możliwe dzięki wykorzystanemu narzędziu zarządzającemu Maven. Badanie objęło wykonanie 10 prób budowania całego projektu oraz wyciągnięcie średniej.

4.4. Objętość API

Kwestią niewątpliwie istotną z punktu widzenia użytkownika jest objętość czyli w gruncie rzeczy możliwości API (ang. application programming interface), z którym przyjdzie mu pracować. Nie jest tajemnicą, że mniej obszerne możliwości danej technologii wymuszają niekiedy na programiście wykonywanie dodatkowej pracy w celu rozwiązania problemu, który jest tak ogólny, powszechny, a czasem nawet trywialny, że logicznym byłoby umieszczenie odpowiedniego API już w samej bibliotece standardowej co się niestety nie zawsze zdarza.

W celu ocenienia obszerności poszczególnych API zdecydowano się na metodę oceny ilościowej, tj. porównano bezpośrednio liczbę bytów klasopodobnych oraz metod, funkcji i pól w nich zawartych. Byt klasopodobny rozumiany jest tutaj jako każda definicja abstrakcyjnego typu lub kontenera jak klasy, interfejsy, cechy (ang. traits), obiekty (singletonowe definicje w Kotlinie i Scali) czy wreszcie paczki mogące być kontenerami na funkcje niebędące przypisanymi do konkretnych klas (funkcje najwyższego poziomu w Kotlinie bądź obiekty paczkowe w Scali). Dodatkowo, wszelkie elementy definiujące wewnętrzną strukturę bytów klasopodobnych, tak jak wcześniej wymienione metody, czy konstruktory, nazwano dalej zbiorczo własnościami.

5. Badania

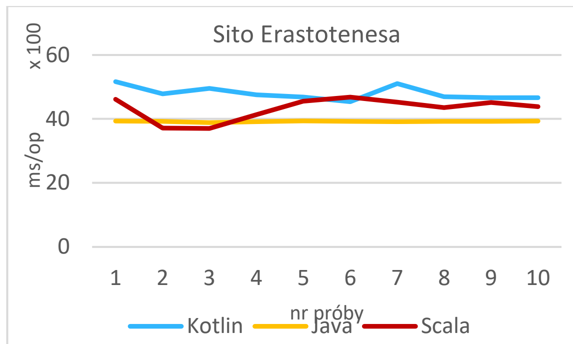
5.1. Wydajność

Wykonany na wszystkich dostępnych implementacjach algorytmu Sita Eratostenesa test nie stwierdził znaczącej różnicy w szybkości wykonania równoległego kodu w implementacjach w poszczególnych językach. Średnia różnica pomiędzy Kotlinem a Javą zamyka się w ok. 18% na niekorzyść tego pierwszego zaś Scala wypadła o ok. 10% gorzej od Javy. Klasę testową dla Kotlina prezentuje Listing 1. zaś szczegóły wszystkich kolejnych prób Rysunek 6.

Listing 1: Fragment klasy testującej znajdowanie liczb pierwszych w Kotlinie

```
@BenchmarkMode(Mode.AverageTime)
@Warmup(iterations = 5, time = 5, timeUnit = TimeU-
nit.SECONDS)
@Measurement(iterations = 10, time = 15, timeUnit =
TimeUnit.SECONDS)
@Fork(1)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
open class SievePrimesBenchmark {

   @Benchmark
   fun findPrimesParallel(blackhole: Blackhole) {
       val primes: List<Int> = runBlock-
ing(Dispatchers.Default) { findPrime-
Numbers(2_100_000_000) }
       blackhole.consume(primes)
   }
}
```



Rysunek 6: Szybkość wykonania metod znajdujących liczby pierwsze w kolejnych próbach

Przed przejściem do następnych testów należy doprecyzować dane, jakie przedstawiają wykresy wydajnościowe. Otóż zarówno powyższy z Rysunek 6, jak i późniejsze, prezentują na skali wartości w formie mikrosekund na jedną operację (ms/op), przy czym operacja definiowana jest jako podstawowa jednostka działania w JMH, oznaczając domyślnie jedno wykonanie metody z adnotacją @Benchmark. Istnieją oczywiście opcje konfiguracyjne pozwalające na zmianę tego zachowania, jednak na potrzeby niniejszego artykułu pozostawiono konfigurację domyślną.

W kolejnym teście bazującym już na algorytmie znajdowania otoczki wypukłej sytuacja jest zgoła odmienna. Implementacja w kotlinowych współprogramach wypadła o ok. 13% lepiej od analogicznej logiki w Javie. Bardzo znaczące jest przy tym, że w wypadku programu javowego zaprezentowany wynik udało się uzyskać dopiero po wielogodzinnych próbach optymalizacyjnych gdyż pierwotny rezultat okazał się być niemal 1,5 razy wolniejszy od implementacji w Kotlinie. Najgorzej poradziła sobie Scala, która ostatecznie oferowała wykonanie o 42% wolniejsze od zwycięzcy próby. Całość wyników prezentuje Rysunek 7., zaś klasę testową Listing 2.

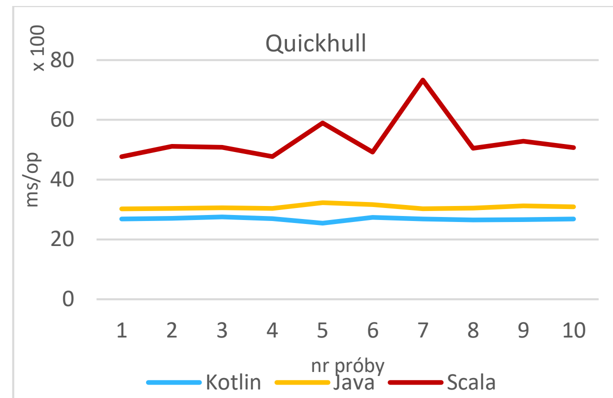
Listing 2: Fragment klasy testująca znajdowanie otoczki wypukłej w Kotlinie.

```
@BenchmarkMode(Mode.AverageTime)
@Warmup(iterations = 5, time = 6, timeUnit = TimeU-
nit.SECONDS)
@Measurement(iterations = 10, time = 20, timeUnit =
TimeUnit.SECONDS)
@Fork(1)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
open class QuickHullBenchmark {

   @State(Scope.Benchmark)
   open class Data {
       // (...) inicjalizacja danych testowych
   }

   @Benchmark
   fun findConvexHullParallel(data: Data, black-
hole: Blackhole) {
       val quickHull: QuickHull = Quick-
Hull(data.points)
       val points: List<Point> = runBlock-
ing(Dispatchers.Default) { quickHull.find() }

       blackhole.consume(points)
   }
}
```



Rysunek 7: Szybkość wykonania metod znajdujących otoczkę wypukłą w kolejnych próbach

W ostatnim przypadku testowym szybkiej transformacji Fouriera wyniki prezentują się nadspodziewanie równo. Świadczy o tym niskie odchylenie standardowe, które dla żadnego z badanych języków nie przekroczyło 2500. Spowodowane może to być charakterystyką stworzonego algorytmu, która po części bazuje na blokującym zadaniu odczytywania kolejnych porcji danych z pliku. Przedstawiony na Rysunek 8. wykres pokazuje szczegółowe wyniki w kolejnych próbach. W kwestii wydajności po raz drugi najlepszy okazał się Kotlin z wynikiem nieco ponad 5% lepszym od Javy zaś zdecydowanie najgorzej poradziła sobie implementacja w Scali, która była średnio o 38% wolniejsza od zwycięzcy. Jedną z możliwych przyczyn takiego stanu rzeczy mogą być np. różnice w działaniu podstawowych instrukcji języka, które w pewnych okolicznościach mogą nie zapewniać optymalnej wydajności [14, 15]. Poniższy Listing 3. prezentuje klasę testującą powyższy przypadek w Kotlinie, zaś Rysunek 8. wyniki w kolejnych próbach.

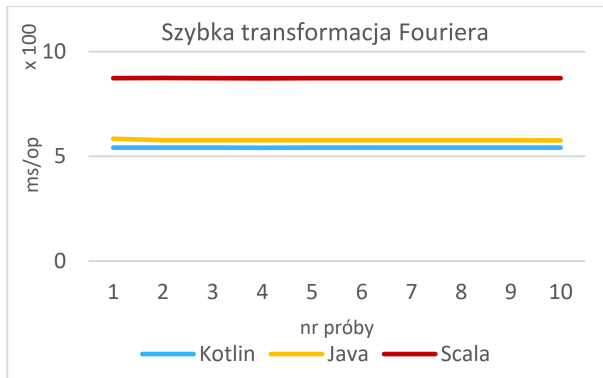
Listing 3: Fragment klasy testującej liczenie FFT.

```
@BenchmarkMode(Mode.AverageTime)
@Warmup(iterations = 5, time = 6, timeUnit = TimeU-
nit.SECONDS)
@Measurement(iterations = 10, time = 20, timeUnit =
TimeUnit.SECONDS)
@Fork(1)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
open class WavFileFftBenchmark {

   @State(Scope.Benchmark)
   open class Data {
       // (...) inicjalizacja danych testowych
   }

   @Benchmark
   fun calculateFftParallel(data: Data, blackhole:
Blackhole) {
       val fftInDecibeles: List<DoubleArray> =
runBlocking(Dispatchers.Default) { trans-
form(data.files) }

       blackhole.consume(fftInDecibeles)
   }
}
```



Rysunek 8: Szybkość wykonania metod liczących FFT w kolejnych próbach

5.2. Długość kodu

Kolejną zmierzoną metryką jest długość kodu. Przy pomiarze kolejnych implementacji wzięto pod uwagę wszystkie klasy wymagane do wykonania zadania założonego dla danego algorytmu, łącznie z testami jednostkowymi sprawdzającymi poprawność implementacji. Wyniki prezentuje Tabela 1.

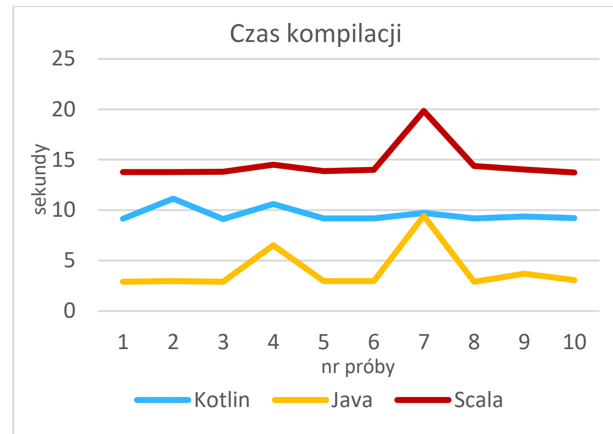
Tabela 1: Pomiary liczby linii kodu poszczególnych implementacji

	Kotlin	Java	Scala
Sito Eratostenesa	108	120	128
Quickhull	149	260	151
FFT	313	334	307
Klasy pomocnicze	21	40	16
Razem	**591**	**754**	**602**

Co łatwo zauważyć po powyższych danych, Java wymaga o ponad 20% więcej kodu do wykonania identycznych zadań co Kotlin lub Scala. Potwierdza to niejako często przytaczany argument o gadatliwości Javy czyli potrzeby pisania nieporównywalnie większej ilości kodu w celu wykonania podobnych zadań co w innych technologiach. Jeśli wejść w szczegóły to Java wypadła stosunkowo najsłabiej przy implementacji Quickhulla oraz klas pomocniczych, które objętościowo zajmowały ponad 50% więcej linii od najlepiej wypadającej w tym punkcie Scali. Całościowo, najlepiej na badanym polu wypadł Kotlin, jednak tuż za nim, z niewiele gorszym wynikiem uplasowała się Scala. Przy wnikliwym porównaniu obu języków uwagę zwraca pomijalnie mała różnica raz na korzyść, a innym razem na niekorzyść Kotlina jednak z jednym wyjątkiem – w przypadku implementacji Sita Eratostenesa Scala wymagała o 15,63% linii kodu więcej niż Kotlin.

5.3. Czas kompilacji

Wykonano 10 prób budowania całego utworzonego na potrzeby artykułu projektu komendą *mvn clean install*. Zadane parametry określają tryby pracy, jakie zdecydowano się włączyć tak, że pierwszy z wymienionych wymusza czyszczenie uprzednio skompilowanych źródeł zaś drugi zleca wykonanie całościowego procesu zbudowania projektu i umieszczenia spakowanego wyniku do lokalnego repozytorium Maven.



Rysunek 9: Czasy kompilacji modułów w kolejnych próbach

Wszystkie pomiary przedstawione zostały na Rysunek 9. Należy tutaj wziąć pod uwagę dosyć niewielki rozmiar projektu, co pokazuje np. badanie długości kodu z podpunktu 4.2., przez co uzyskane wyniki mogą nie być całkowicie reprezentatywne przy rozpatrywaniu możliwości użycia jednej bądź drugiej technologii w dużych aplikacjach jednak same uzyskane wyniki przedstawiają pewien trend, który niewątpliwie mniej lub bardziej utrzyma się nawet w wysokoskalowalnych projektach. Najniższy czas, a zarazem najlepszy wynik uzyskała Java, zaś moduły pozostałych języków budowały się nawet od trzech do czterech razy dłużej.

5.4. Objętość API

Obszerność interfejsów programistycznych do przetwarzania równoległego w wykorzystanych językach była kolejnym kryterium badawczym. Analiza wykonana została dla bytów zawartych w następujących paczkach bibliotek standardowych:

- kotlin.coroutines i kotlinx.coroutines dla Kotlina;
- java.util.concurrent dla Javy;
- scala.concurrent dla Scali.

Dodatkowo, aby nie zaciemniać wyników, pominięto przypadki bytów klasopodobnych, metod, pól oznaczonych jako przestarzałe bądź do usunięcia (poprzez adnotację @Deprecated dla Javy oraz odpowiedników w pozostałych językach) oraz klas niebędących bezpośrednio związanymi z omawianymi mechanizmami, np. w przypadku paczki javowej wyróżnić można wiele implementacji kolekcji dedykowanych do bezpiecznego przetwarzania równoległego, które jednak nie odnoszą się bezpośrednio do badanego obszaru.

Tabela 2: Wyniki obszerności API

	Kotlin	Java	Scala
Byty klasopodobne	52	52	18
Własności	307	415	68

Tabela 2. zawierająca wyniki przeprowadzonej analizy, pokazuje niekwestionowaną przepaść pomiędzy Scalą o dwoma pozostałymi językami. Różnica w przypadku bytów klasopodobnych była ponad dwukrotna, a dla własności wyniosła aż 451% na niekorzyść Scali w porównaniu z drugim wynikiem uzyskanym przez współprogramy Kotlina. Najbardziej obszernym API

może poszczycić się Java, notując zauważalnie wyższą liczbę własności od drugiej z kolei technologii. Jeżeli jednak chodzi o byty klasopodobne to na tym poziomie wypadła równie dobrze co Kotlin.

6. Wnioski

Przeprowadzona analiza porównawcza nie pozwala na jednoznaczne wyłonienie technologii najlepszej do programowania równoległego. Każda z przedstawionych mechanik ma swoje jasne i ciemne strony, tj. w niektórych przypadkach prezentuje się lepiej na tle konkurencji, a w innych gorzej.

Najgorzej w połowie z założonych wcześniej kryteriów porównawczych zachowała się Scala. Słaba na tle konkurentów dokumentacja oraz co najwyżej zbliżona do nich wydajność, wskazują, że język ten jest raczej w odwrocie i sprawdza się dobrze jedynie w bardzo specyficznych niszach. Zaletą współbieżnej Scali jest za to niewątpliwie zwięzłość tworzonych rozwiązań, będąca na zbliżonym poziomie co w Kotlinie i jednocześnie zdecydowanie lepsza niż w Javie.

Z kolei współbieżny interfejs programistyczny Javy zachował się dobrze lub poprawnie w kryteriach takich jak obszerność API czy czas kompilacji. Zdecydowanie gorzej zaprezentował się przy dokumentacji online, długości kodu wynikowego, możliwościach debuggingu oraz podatności na błędy. Wynika to przynajmniej częściowo z długiego stażu tej technologii na rynku i przekłada się na bycie odpowiednim wyborem dla dużych i długookresowych projektów, w których zaangażowane jest znaczne grono osób.

Tytułowe współprogramy Kotlina poradziły sobie w przeprowadzonych badaniach nierówno, tj. w kwestii zwięzłości kodu okazały się prezentować najlepsze wyniki. Idąc dalej, odznaczały się podobną lub nawet lepszą wydajnością od Javy i podobnie rozległym API. Zdecydowanie najsłabiej wyglądało tu kryterium czasu kompilacji, w którym wynik był kilka razy gorszy od Javy i na podobnym poziomie co Scala. Nie jest jednak pewne czy podobna różnica zaistniałaby również w przypadku projektu o dużej skali jako, że stworzone na potrzeby pracy rozwiązanie nie jest bardzo rozległe i z pewnością nie oddaje w 100% przypadków użycia mogących wystąpić w biznesie czy nauce. Wymienione wyżej cechy wskazują na idealne dopasowanie Kotlina do nowych projektów realizowanych, np. przez startupy lub mniejsze firmy oraz, dzięki niskiej podatności na błędy, dla zastosowań akademickich. Krótki okres obecności na rynku, a co za tym idzie mniejsza stabilność zdefiniowanego API czy wsparcia od stron trzecich skłania do wniosku, że w przypadku np. wielkoskalowych projektów o długim czasie wsparcia, ostateczny wybór powinien być poprzedzony dodatkowymi analizami z innych punktów widzenia.

Literatura

[1] Parallel Computing: Background, https://www.intel.com/pressroom/kits/upcrc/ParallelComputing_backgrounder.pdf, [22.05.2020].

[2] M. E. Conway, Design of a Separable Transition Diagram Compiler, Communications of the ACM 6.7 (1963) 396-408.

[3] A. L. De Moura, R. Ierusalimschy, Revisiting Coroutines, ACM Transactions on Programming Languages and Systems 31.2 (2009) 1-31.

[4] D. Racordon D, Coroutines with Higher Order Functions, arXiv preprint arXiv:1812.08278 (2018).

[5] A. Ohlsson, E. Leffler E, A Coroutine Extension to Java, (2018).

[6] D. E. Damasceno Costa, C. Bezemer, P. Leitner, A. Andrzejak, What's Wrong With My Benchmark Results? Studying Bad Practices in JMH Benchmarks, IEEE Transactions on Software Engineering (2019).

[7] M. E. O'Neill, The Genuine Sieve of Eratosthenes, Journal of Functional Programming 19.1 (2009) 95-106.

[8] Sieve of Eratostheneses animation, https://en.wikipedia.org/wiki/File:Sieve_of_Eratosthenes_animation.gif, [22.05.2020].

[9] J. S. Greenfield, A Proof for a QuickHull Algorithm (1990).

[10] Quickhull example, https://en.wikipedia.org/wiki/File:Quickhull_example3.svg, [22.05.2020].

[11] Quickhull example, https://en.wikipedia.org/wiki/File:Quickhull_example6.svg, [22.05.2020]..

[12] Quickhull example, https://en.wikipedia.org/wiki/File:Quickhull_example7.svg, [22.05.2020]

[13] D. Nam, A. Horvath, A. Macvean, B. Myers, B. Vasilescu, MARBLE: Mining for Boilerplate Code to Identify API Usability Problems, 2019 34th IEEE/ACM International Conference on Automated Software Engineering (2019) 615-627.

[14] Why are Scala for loops slower than logically identical while loops?, https://stackoverflow.com/questions/21373514/why-are-scala-for-loops-slower-than-logically-identical-while-loops, [22.05.2020].

[15] Micro-optimizing your Scala code, https://www.lihaoyi.com/post/MicrooptimizingyourScalacode.html#speed-through-while-loops, [22.05.2020].