PRACE NAUKOWE Akademii im. Jana Długosza w Częstochowie

Technika, Informatyka, Inżynieria Bezpieczeństwa 2017, t. V, s. 25–38

http://dx.doi.org/10.16926/tiib.2017.05.02

Jerzy Krawiec
Politechnika Warszawska
Plac Politechniki 1, 00-661 Warszawa
e-mail: j.krawiec@wip.pw.edu.pl

WPŁYW PROGRAMOWANIA RÓWNOLEGŁEGO NA WYDAJNOŚĆ PROGRAMU JAVY

Streszczenie. W artykule przedstawiono praktyczne aspekty programowania obiektowego w języku Java w zakresie programowania równoległego, czyli techniki stosowanej w celu wykorzystywania komputerów wieloprocesorowych (lub wielordzeniowych). Przedstawiono mechanizmy, które zapewniają programistom Javy korzystanie z wielu procesorów w przejrzysty i skalowany sposób. Zaprezentowano mechanizmy wspierające techniki programowania równoległego. Przedstawiono technikę rekurencji w ramach strategii „Dziel i zwyciężaj" oraz zasady przetwarzania sekwencyjnego. Zbadano możliwości zwiększenia kodu programu Javy w zakresie technik programowania równoległego na przykładzie frameworku *Fork/Join*. Przedstawiono możliwości tego frameworku pod kątem zwiększenia wydajności programu Javy. Przeprowadzono pomiary czasu wykonania programu dla różnych poziomów równoległości oraz różnych progów przetwarzania sekwencyjnego. Wykazano, że odpowiednia konstrukcja kodu Javy może znacznie skrócić czas wykonywania programu Javy, co przekłada się na wydajność programu.
Słowa kluczowe: programowanie, programowanie równoległe, Java, framework, wydajność programu, czas wykonania programu.

THE IMPACT OF PARALLEL PROGRAMMING ON PERFORMANCE OF JAVA PROGRAM

Abstract. The article presents the practical aspects of object-oriented programming language Java in the field of parallel programming, a technique used in order to use multiprocessor computers (or multi-core). Mechanisms supporting parallel programming techniques were presented. Recursion technique in the framework of the "Divide and conquer" and the principle of sequential processing were presented. We examined the possibility of increasing the Java code program in the field of parallel programming

on the example framework Fork/Join. The possibilities of this framework to improve performance of a Java program were presented. The measurements of the program runtime for different levels of parallelism and different thresholds for sequential processing were conducted. It has been shown that proper design of Java code can significantly shorten the duration of the program Java, which translates to program performance.
Keywords: Programming, parallel programming, Java, code efficiency, framework, runtime.

Wprowadzenie

Większość nowoczesnych języków programowania jest kompilowana do kodu wykonywalnego, co zapewnia większą wydajność systemu. Jednakże w Javie wyjściem generowanym przez kompilator języka jest interpreter kodu bajtowego, zwany maszyną wirtualną Javy (JVM[1]). Kod bajtowy to zoptymalizowany zestaw instrukcji. Interpretacja programu do kodu bajtowego znacznie ułatwia uruchomienie programu na wielu platformach (środowiskach), przez co uzyskuje się efekt przenośności – jedną z najważniejszych cech Internetu. Przenośność to możliwość uruchomienia programu na maszynie o dowolnej liczbie procesorów/rdzeni. Takiej przenośności nie można uzyskać w przypadku kodu wykonywalnego [1].

Ogólnie rzecz ujmując, program skompilowany do postaci pośredniej (bajtowej) i interpretowany przez JVM działa wolniej niż program skompilowany do postaci wykonywalnej. Jednakże różnica w wydajności może okazać się nieznacząca, jeżeli kod programu jest odpowiednio zapisany.

Powszechnie obecnie stosowane formy wykonania programu są określane mianem przetwarzania współbieżnego, równoległego i rozproszonego. Programem może być system operacyjny, kompilator języka programowania, maszyna wirtualna, np. JVM, lub program użytkowy. Wykonanie programu to realizacja określonych rozkazów i instrukcji, których celem jest rozwiązanie określonego problemu obliczeniowego. Rozkazy to pojedyncze polecenia realizowane przez procesory, które nie są bezpośrednio związane z kodem programu. Zbiór rozkazów dotyczy konkretnego uruchomienia programu. Konkretna czynność związana z wykonaniem programu jest przypisana danemu rozkazowi determinowanemu przez dane wejściowe programu. Natomiast instrukcje wykonują bardziej złożone zadania definiowane w postaci kodu i tłumaczone na rozkazy procesora. Zbiór rozkazów z reguły jest dzielony na podzbiory, gdzie rozkazy są wykonywane w sposób sekwencyjny, czyli w kolejności zdefiniowanej przez pro-

---

[1] Java Virtual Machine.

gramistę lub według kolejności określanej przez kompilator kodu źródłowego. Takie wykonanie kodu programu nazywane jest wątkiem.

Przetwarzanie współbieżne to sytuacja, w której rozkazy jednego wątku zaczynają się wykonywać już w momencie, gdy poprzedni wątek nie jest jeszcze zakończony. Jeżeli przynajmniej dwa rozkazy wątków wykonywanych współbieżnie są realizowane w tym samym czasie, to wtedy mamy do czynienia z przetwarzaniem równoległym. Przetwarzanie równoległe wymaga specjalistycznego sprzętu, np. maszyny z wieloma procesorami (rdzeniami) zarządzanej przez jeden system operacyjny, ale może być to także sieć komputerów, które mają swój system operacyjny. Uruchamianie współbieżne jest stosowane od wielu lat, program jest uruchamiany współbieżnie z innymi programami. Jest to zasługą przede wszystkim powszechnie stosowanych wielozadaniowych systemów operacyjnych. Program jest oddzielnym zbiorem rozkazów, które są wykonywane współbieżnie przez system operacyjny. W jednej sekundzie procesor wykonuje setki tysięcy operacji wejścia-wyjścia, ale przez znaczną część czasu jest w stanie uśpienia. Zatem przetwarzanie współbieżne jest pojęciem bardziej ogólnym w stosunku do przetwarzania równoległego. W związku z tym każde przetwarzanie równoległe jest przetwarzaniem współbieżnym, ale nie każde przetwarzanie współbieżne jest przetwarzaniem równoległym [2].

Kluczowymi elementami w programowaniu równoległym jest zidentyfikowanie współbieżności podczas realizacji programu. Współbieżności mogą mieć charakter twórczy lub techniczny. Zdefiniowanie synchronizacji pomiędzy procesami oznacza, że model programowania jest znany. W tym aspekcie zapewnienie przenośności programu ma zasadnicze znaczenie.

Architektura wieloprocesorowa, a ściślej dostęp do pamięci operacyjnej przez poszczególne procesory, wpływa na tzw. zrównoleglanie programu. Pamięć współdzielona ułatwia programowanie i wpływa na wydajność programu, ponieważ umożliwia lepszą komunikację pomiędzy procesorami. Jednak należy pamiętać, że liczba procesorów, które mogą być w ten sposób połączone, jest ograniczona do kilkudziesięciu. Architektura bazująca na pamięci rozproszonej umożliwia łączenie znacznie większej liczby procesorów, rzędu kilku tysięcy, a koszty związane z jej zastosowaniem są znacznie niższe niż architektury bazującej na pamięci współdzielonej. Trzeba jednak liczyć się z tym, że programowanie staje się trudniejsze, a komunikowanie się między procesorami zachodzi wolniej w stosunku do architektury bazującej na pamięci współdzielonej [3].

Jednym z powszechnie stosowanych sposobów tworzenia programów równoległych jest zdefiniowanie metodologii. W jej ramy należałoby ująć wszystkie aspekty programowania równoległego. Taka metodologia zakłada pięć podstawowych zadań, które powinny być wykonane:

- dekompozycja zadań na podzadania;
- odwzorowanie zadań na procesy i wątki oraz elementy przetwarzania;

- podzielenie danych na elementy pamięci dotyczące elementów przetwarzania;
- zdefiniowanie sposobu wymiany danych między procesami oraz odzwierciedlenie go w sieci komunikacji między procesorami (rdzeniami);
- określenie sposobu synchronizacji procesów.

Specyficznym rodzajem stosowanej metodologii jest metodologia PCAM[2], która zakłada dzielenie na zadania, określa sposób komunikacji oraz podział danych na wspólne i prywatne, określa sposób korzystania z danych, zapewnia synchronizację operacji na danych, analizę wariantów podziału oraz uwzględnia odwzorowanie programu na architekturę sprzętową [5]. Dość istotnym pojęciem w programowaniu równoległym jest równoległość danych, które oznacza model programowania, który definiuje sposób przydziału danych poszczególnym procesom oraz określa operacje na danych. Sama realizacja programu uwzględniająca komunikację i synchronizację procesów jest wykonywana przez środowisko (JVM) bez jawnego udziału programisty. W bardziej ogólnym (szerszym) sensie równoległość danych jest modelem tzw. zrównoleglenia danych poprzez dekompozycję danych, przydzielenie danych procesom i dopiero wykonanie programu zgodnie z jego twórcą. Model wykonania równoległego może być także rozważany jako model dekompozycji danych, gdzie na podstawie indeksu procesor ustala dane, na których operuje. Alternatywą takiego modelu jest model bazujący na równoległości sterowania, w którym na podstawie indeksu procesor określa ścieżkę realizacji programu lub iterację pętli.

Pomiędzy procesami powinna być zapewniona synchronizacja (uporządkowanie w czasie) i komunikacja (przesyłanie danych). Komunikacja polega na przesyłaniu do procesów (wątków) odpowiednich komunikatów, tzw. tokenów. Proces przesyłania może mieć charakter synchroniczny i asynchroniczny. Natomiast synchronizacja polega na zapewnieniu odpowiednich zależności czasowych w odniesieniu do procesów (wątków). Synchronizacja jest konieczna, gdy procesy mają wspólne struktury danych lub zapewniają dostęp do wspólnych zasobów [4].

Spełnienie wymagań programowania współbieżnego jest realizowane powszechnie za pomocą interfejsu Concurrent API[3]. Chociaż oryginalny interfejs był dość mocno rozbudowany i miał duże możliwości w zakresie tworzenia aplikacji wielowątkowych, to dopiero wprowadzenie frameworku *Fork/Join* umożliwia efektywne działanie programów w systemach wielordzeniowych [1]. Framework zapewnia rzeczywistą współbieżność, a nie sztuczny podział czasu procesora, co wprost przekłada się na czas wykonywania programu, a zatem i na jego wydajność.

---

[2] Partitioning Communication Agglomeration Mapping.
[3] Application Programming Interface.

Framework *Fork/Join* to mechanizm w postaci nowych klas i interfejsów wspierających programowanie równoległe. Framework został zdefiniowany w pakiecie *java.util.concurrent*. *Fork/Join* umożliwia wykorzystanie programowania wielowątkowego w dwojaki sposób:

- upraszcza tworzenie i stosowanie wielu wątków;
- automatycznie dostosowuje działanie do liczby dostępnych procesorów.

W systemach jednordzeniowych mechanizm wielowątkowości polega na współdzieleniu czasu procesora na poszczególne zadania, ale nie zapewnia optymalizacji przy wykorzystaniu systemów wielordzeniowych. W systemach wieloprocesorowych istnieje możliwość korzystania poszczególnych części kodu programowego z kilku procesorów jednocześnie. Takie środowisko umożliwia przyśpieszenie niektórych operacji. Najważniejsze klasy frameworku to:

- *ForkJoinTask<V>*;
- *ForkJoinPool*;
- *RecursiveAction*;
- *RecursiveTask<V>*.

Klasa *ForkJoinTask<V>* to klasa abstrakcyjna, które definiuje zadanie i może być zarządzana przez obiekt typu *ForkJoinPool*. Parametr *V* określa typ danych zwracanych przez zadanie. Zadania typu *ForkJoinTask* są realizowane przez wątki, zarządzane przez pulę wątków typu *ForkJoinPool*. Zadania typu *ForkJoinTask* są bardziej efektywne niż wątki. Do najważniejszych metod definiowanych przez klasę *ForkJoinTask<V>* należą:

- *final ForkJoinTask<V> fork()* – zwraca *this* po zaplanowaniu wykonania odpowiedniego zadania;
- *final V join()* – zwraca wynik zadania po jego zakończeniu, dla którego została wywołana;
- *final V invoke()* – inicjuje nowe zadanie, a po jego zakończeniu zwraca jego wynik.

Za pomocą metod *fork()* i *join()* można inicjować wiele zadań oraz wywoływać wiele zadań jednocześnie (metoda *invoke()*).

Klasa *ForkJoinPool* to klasa służąca do zarządzania zadań typu *ForkJoinTask*. Klasa definiuje następujące, najczęściej stosowane konstruktory [6]:

- *ForkJoinPool()* – domyślna pula zadań;
- *ForkJoinPool(int poziom)* – dodatkowo umożliwia określenie poziomu równoległości.

Do zarządzania wątków klasa *ForkJoinPool* stosuje mechanizm zwany wykradaniem zadań. Każdy wątek jest przypisany do kolejki zdarzeń. Jeżeli kolejka zdarzeń jakiegoś wątku jest pusta, zadanie jest odbierane innemu wątkowi. Taki mechanizm umożliwia zwiększenie wydajności programu i zapewnia zrównoważone obciążenia.

Klasa *RecursiveAction* jest podklasą *ForkJoinTask<V>* i reprezentuje zadanie, które nie zwraca żadnego wyniku. *RecursiveAction* definiuje cztery metody, ale najczęściej stosuje się metodę *compute()*, która reprezentuje część obliczeniową zadania. Metoda ta jest definiowana następująco: *protected abstract void compute().*

Klasa *RecursiveAction* jest wykorzystywana do implementacji strategii „Dziel i zwyciężaj" do zadań, które nie zwracają żadnego wyniku.

Klasa *RecursiveTask<V>* jest podklasą *ForkJoinTask<V>* i reprezentuje zadania, które zwracają wynik. Typ wyniku jest reprezentowany przez parametr *V*. Klasa ta definiuje najczęściej stosowaną metodę *compute()* w następujący sposób: *protected abstract V compute().*
Klasa *RecursiveTask<V>* jest wykorzystywana do implementacji strategii „Dziel i zwyciężaj" do zadań, które zwracają jakiś wynik.

Strategia „Dziel i zwyciężaj" stosuje technikę rekurencji, polegającą na rekurencyjnym dzieleniu zadań na podzadania tak małe, aby można je było wykonać w trybie sekwencyjnym. Proces dzielenia zadania trwa dopóty, dopóki nie uzyskamy wartości progowej, od której przetwarzanie sekwencyjne zajmuje mniej czasu niż dalsze dzielenie zadań. Atutem tej strategii jest stworzenie możliwości równoległego przetwarzania danych. Kluczowym warunkiem efektywności strategii „Dziel i zwyciężaj" jest odpowiedni wybór progu, który stwarza możliwość przetwarzania sekwencyjnego. Optymalizacja progu przetwarzania sekwencyjnego zależy od czasu niezbędnego w celu przetworzenia danych. Dokumentacja API Javy rekomenduje od 100 do 10 000 kroków obliczeniowych dla pojedynczego zadania. Należy pamiętać, że oprócz aplikacji w Javie, również system operacyjny musi mieć dostęp do czasu procesorów, co oznacza, że program nie będzie miał nieograniczonego dostępu do wszystkich procesorów.

Poziom równoległości to liczba wątków, które mogą być wykonywane współbieżnie, co oznacza liczbę zadań realizowanych jednocześnie z zastrzeżeniem, że liczba tych zadań nie może być większa niż liczba procesorów. Domyślny konstruktor klasy *ForkJoinPool* oznacza, że program używa domyślnego poziomu równoległości odpowiadającej liczbie procesorów. Poziom równoległości może być określany poprzez utworzenie obiektu klasy *ForkJoinPool* według następującej wersji konstruktora: *ForkJoinPool(int poziom).* Wartość parametru *poziom* określa poziom równoległości, który musi zawierać się w zakresie od 1 do wartości mniejszej niż wartość graniczna danej implementacji.

Do pomiarów czasu wykonywania programu najczęściej stosuje się metodę *nanoTime()* z klasy *System*. Metoda zwraca wynik z zegara z dokładnością rzędu nanosekund. W celu ułatwienia eksperymentów można wykorzystać następujące metody:

- *int getParallelism()* – metoda zdefiniowana w klasie *ForkJoinPool*;
- *int availableProcessors()* – metoda zdefiniowana w klasie *Runtime*.

Metoda *getParallelism()* jest stosowana do określenia bieżącego poziomu równoległości – domyślnie równego liczbie dostępnych procesorów.

Za pomocą metody *availableProcessors()* można określić liczbę dostępnych procesorów w systemie. Wynik zwracany przez tę metodę może być za każdym razem inny ze względu na żądania pozostałych procesorów pracujących w danym systemie.

Badania wydajności programu Javy

Wszystkie pomiary przeprowadzono za pomocą komputera o następujących parametrach:

- procesor: Intel(R) Core™ i5-2520M CPU @ 2,5 GHz;
- pamięć RAM: 4GB;
- dysk: 112GB SSD;
- karta graficzna: Intel (R) HD Graphics 3000;
- system operacyjny: WINDOWS 7 64-bitowy;
- Java Platform (JDK) 8u73 / 8u74.

Wydajność programu jest funkcją czasu wykonania operacji wg następującego wzoru:

$$W = \frac{1}{t}$$

w którym:
W – wydajność kodu programu;
t – czas wykonania programu.
Różne wartości czasu wykonywania operacji są wynikiem wahań wydajności procesora, która jest trudna do ustalenia w momencie pomiaru. Dlatego też obliczano wartość średnią czasu wykonania operacji na podstawie 100 pomiarów, a w tabeli wyników uwzględniono 10 najmniejszych wartości czasu wykonania programu każdego wariantu. Do pomiaru czasu wykonania programu służącego do tworzenia zadania frameworku *Fork/Join* przypisano pierwiastek sześcienny z liczb, będących elementami tablicy.

W badaniach użyto następującego kodu źródłowego (badania przeprowadzono dla poziomów równoległości 1 i 2 oraz progów przetwarzania sekwencyjnego: 100, 300, 700, 1 000, 3 000, 7 000, 10 000):

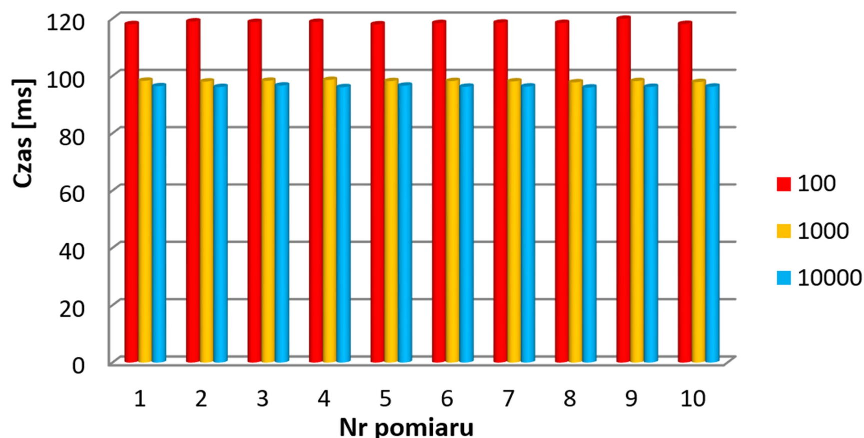
```
import java.util.concurrent.*;
class Tablica extends RecursiveAction {
 int progi_sekw;
 double[] dane;
```

```
int start, end;
Tablica(double[] abc, int a, int b, int p ) {
dane = abc;
start = a;
end = b;
progi_sekw= p;
}
@Override
protected void compute() {
if((end - start) < progi_sekw) {
for(int i = start; i < end; i++) {
dane[i] = Math.cbrt(dane[i]);
}
}
else {
int middle = (start + end) / 2;
invokeAll(new Tablica(dane, start, middle, progi_sekw),
new Tablica(dane, middle, end, progi_sekw));
}
}
}
class Framework {
public static void main(String jekr[]) {
int poziom;
int progi;
if(jekr.length != 2) {
System.out.println("Sposób użycia: FJExperiment poziom-równoległości próg ");
return;
}
poziom = Integer.parseInt(jekr[0]);
progi = Integer.parseInt(jekr[1]);
long beginTime, endTime;
ForkJoinPool pula = new ForkJoinPool(poziom);
```

```
double[] liczba = new double[1000000];
for(int i = 0; i < liczba.length; i++)
liczba[i] = (double) i;
Tablica zadanie = new Tablica(liczba, 0, liczba.length, progi);
beginTime = System.nanoTime();
pula.invoke(zadanie);
endTime = System.nanoTime();
System.out.println("Poziom równoległości: " + poziom);
System.out.println("Próg przetwarzania sekwencyjnego: " + progi);
System.out.println("Czas działania: " + (endTime - beginTime) + " ns");
System.out.println();
}
```

W tabeli 1 przedstawiono pomiary czasów wykonania programu do tworzenia zadania frameworku *Fork/Join* w zależności od progu przetwarzania równoległego przy poziomie równoległości 1.

Rysunek 1 przedstawia wykres czasu wykonania programu (oś rzędnych) dla 10 pomiarów (oś odciętych) w przypadku różnych progów przetwarzania: 100, 1 000 i 10 000 przy poziomie równoległości 1.



Rys. 1. Porównanie czasów wykonywania programu do tworzenia zadania frameworku *Fork/Join* przy poziomie równoległości 1
Źródło: opracowanie własne.

W tabeli 2 przedstawiono wyniki pomiaru dla tego samego programu, ale przy poziomie równoległości 2.

Tab. 1. Pomiary czasów wykonywania programu [ns] do tworzenia zadania frameworku *Fork/Join* przy poziomie równoległości 1

Próg przetwarzania	Nr pomiaru										X_0
	1	2	3	4	5	6	7	8	9	10	
100	118048383	119002852	118797179	118832074	117987625	118437971	118592328	118469581	119928585	118126793	118622337
300	105795839	107723229	107966257	105060188	107450644	105115609	105666936	107275762	106507681	105721536	106428368
700	100627808	100515325	100825678	98152785	98565357	98108038	99802253	99987808	98721355	100316634	99562304
1 000	98309542	98038596	98299689	98607583	98188848	98206501	98084574	97762723	98201985	97849754	98154980
3 000	97465965	97820674	98122406	97761560	97853105	98136364	97167127	98053439	96725818	97116633	97622309
7 000	97379776	96892900	97062034	96492643	96696672	96722534	96915889	96083766	96375645	97097749	96771961
10 000	96386234	96123910	96635423	96065205	96597654	96218741	96290993	95945741	96164962	96258561	96268742

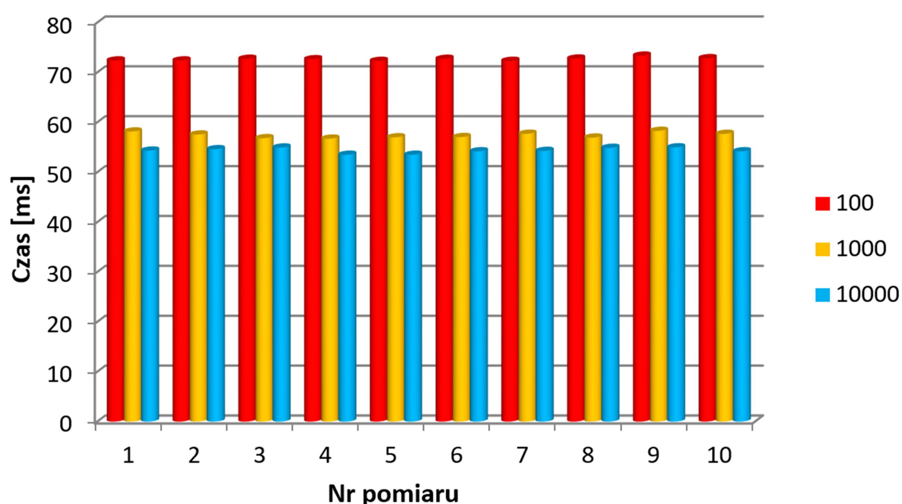
X_0 – wartość średnia
Źródło: opracowanie własne.

Tab. 2. Pomiary czasów wykonywania programu [ns] do tworzenia zadania frameworku *Fork/Join* przy poziomie równoległości 2

Próg przetwa-rzania	Nr pomiaru										X_0
	1	2	3	4	5	6	7	8	9	10	
100	72389803	72392226	72726019	72642273	72305647	72716987	72310983	72769124	73357398	72845480	72645594
300	67709337	68942867	69098404	67238520	68768412	67273990	67626839	68656488	68164916	67661783	68114156
700	66414353	66340115	66544947	64780838	65053136	64751305	65869487	65991953	65156094	66208978	65711121
1 000	58152161	57546234	56813456	56715342	56985053	57034316	57687863	56934560	58285580	57671442	57382601
3 000	56530260	56735991	56910995	56701705	56754801	56919091	56356934	56870995	56100974	56327647	56620939
7 000	55506472	55228953	55325359	55000807	55117103	55131844	55242057	54767747	54934118	55345717	55160018
10 000	54314212	54599523	54938613	53484553	53482910	54177920	54259613	54871698	54969402	54177099	54327554

Źródło: opracowanie własne.

Rysunek 2 przedstawia wykres czasu wykonania programu (oś rzędnych) dla 10 pomiarów (oś odciętych) w przypadku różnych progów przetwarzania: 100, 1 000 i 10 000 przy poziomie równoległości 2.



Rys. 2. Porównanie czasów wykonywania programu do tworzenia zadania frameworku *Fork/Join* przy poziomie równoległości 2
Źródło: opracowanie własne.

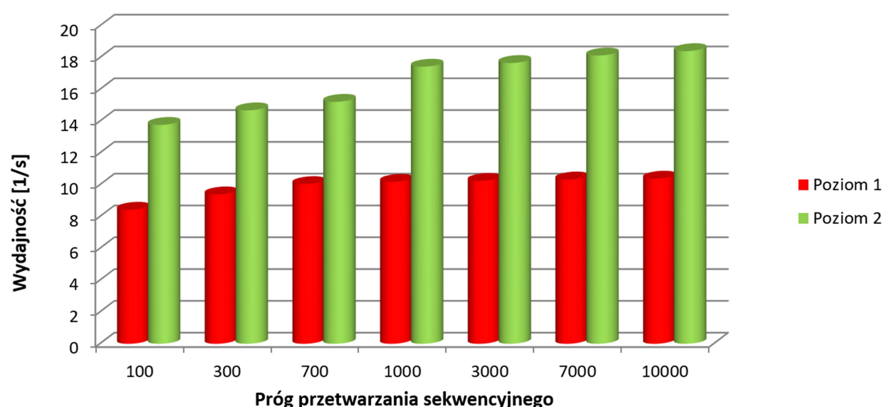
W tabeli 3 przedstawiono wyniki obliczeń wydajności programu na podstawie pomiarów czasu wykonania programu do tworzenia zadania frameworku *Fork/Join* w zależności od progów przetwarzania sekwencyjnego i poziomów równoległości.

Tab. 3. Wyniki obliczeń wydajności programu [1/s] do tworzenia zadania frameworku *Fork/Join*

Poziom równoległości	**Próg przetwarzania sekwencyjnego**						
	100	300	700	1 000	3 000	7 000	10 000
1	8,43	9,40	10,04	10,19	10,24	10,33	10,39
2	13,77	14,68	15,22	17,43	17,66	18,13	18,41

Źródło: opracowanie własne.

Na rysunku 3 przedstawiono porównanie wydajności programu (oś rzędnych) dla 10 pomiarów (oś odciętych) w przypadku progów przetwarzania sekwencyjnego z zakresu (100–10 000) i poziomów równoległości 1 i 2.



Rys. 3. Porównanie wydajności programu w zależności od progów przetwarzania sekwencyjnego dla różnych poziomów równoległości
Źródło: opracowanie własne.

Podsumowanie

Programowanie równoległe i współbieżne umożliwia efektywne gospodarowanie mocą na poziomie fizycznym (sprzętowym). Przetwarzanie równoległe znacznie skraca czas wykonywania obliczeń niż w przypadku przetwarzania sekwencyjnego. Natomiast przetwarzanie współbieżne, również w środowisku jednoprocesorowym, umożliwia dekompozycję zadań, czyli np. niezależne przetwarzanie danych, operacje wejścia-wyjścia oraz komunikowanie się z użytkownikiem.

Odpowiednio zapisany kod programu ma duży wpływ na wydajność programu Javy. Konstrukcja kodu Javy może znacznie skrócić czas wykonywania programu Javy. Nieznaczne wahania czasu wykonania operacji dla różnych pomiarów wynikają z różnej wydajności samego procesora przy różnych pomiarach.

Przedstawione badania dowodzą, że zwiększenie wydajności programu może być uzyskane poprzez większą wartość poziomu równoległości oraz odpowiednio dobranego do nich progu przetwarzania sekwencyjnego. Z przeprowadzonych badań wprost wynika, że zwiększenie progu przetwarzania sekwencyjnego ze 100 do 1 000 skutkuje ponad 20-procentowym wzrostem wydajności programu (21% z jednym procesorem i 27% z dwoma procesorami), ale zwiększenie progu przetwarzania sekwencyjnego z 1 000 do 10 000 powoduje tylko nieznaczny wzrost wydajności programu (2% i 6% odpowiednio). Wzrost wydajności programu z procesorem dwurdzeniowym w stosunku do jednordzeniowego, dla zakresu progów przetwarzania sekwencyjnego od 100 do 10 000, wynosi od 63% (próg 100) do 77% (próg 10 000).

Literatura

[1] Schildt H., *Java The complete Reference, 9th Edition*, McGraw-Hill Companies, Inc. 2014.
[2] Banaś K., *Programowanie równoległe i rozproszone*, Kraków 2011.
[3] https://www.icm.edu.pl/kdm/Programowanie_równoległe, (data dostępu: 2016-08-17).
[4] Wilczewski M., *Programowanie współbieżne i równoległe,* http://doc player.pl/5020048-Programowanie-wspolbiezne-i-rownolegle-dr-inz-marcin-wilczewski-2013.html, (data dostępu: 2016-08-17).
[5] *Designing Parallel Algorithms*, https://www.mcs.anl.gov/~itf/dbpp/text/node14.html (data dostępu: 2016-08-19).
[6] Oaks S., *Java Performance – The Definite Guide*, O'Reilly Media Inc. 2015.