JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 5 (2017) 142-147
Wysłane: 2017-06-18
Przyjęte: 2017-06-21

Porównanie wydajności różnych wersji Javy

Maciasz Mateusz*
Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie. W niniejszym artykule opisano wyniki badań wydajności dwóch wersji Javy - 1.7 i 1.8. Do tego celu zostały stworzone trzy aplikacje. Pierwsza z nich jest odpowiedzialna za przygotowanie danych do testów. Dwie kolejne implementują testy wydajnościowe w zależności od wersji wirtualnej maszyny Javy. Stworzone metody w aplikacjach testowych, miały za zadanie przeanalizować wydajność operacji na kolekcjach oraz szybkości zmiany wartości zmiennych przez dwa wątki.

Słowa kluczowe: Java; Wydajność; Programowanie; Optymalizacja

*Autor do korespondencji.

Adres/adresy e-mail: mateuszmaciasz92@gmail.com

Performance comparison of different Java versions

Maciasz Mateusz*
Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. This article describes the performance results of two versions of Java - 1.7 and 1.8. Three applications were created for this purpose. The first one is responsible for preparing the test data. Two more implement performance tests depending on the virtual machine version of Java. The methods in the test applications were designed to analyze the performance of collections and the rate of change of the variables by two threads.

Keywords: Java; Performance; Programming; Optimization

*Corresponding author.

E-mail address/addresses: mateuszmaciasz92@gmail.com

1. Wstęp

W dzisiejszych czasach niewiele osób zwraca uwagę na optymalizację, ze względu na szybkość rozwoju technologii i coraz większe zapasy pamięci oraz bardziej wydajne procesory, co sprawia że problem wydajności staje się coraz bardziej realny i namacalny.

Dynamiczny rozwój *Javy* w ostatnich latach zmienił podejście do sposobu programowania w tym języku. Od marca 2014 [1] dostępna jest wersja JDK 1.8 wprowadzająca całkowicie inną koncepcję programowania w porównaniu do dotychczasowej. Język ten stał się bardziej funkcyjny niż dotychczas. Został także rozszerzony o możliwość prostszego programowania równoległego. Nowe API [1] wprowadza dużo uproszczeń i zmian oraz kilka optymalizacji do już istniejących metod. Jedną z podstawowych nowości są wyrażenia *lambda* dostępne dotychczas w innych językach programowania. Zostały także dodane strumienie - rozumiane jako widoki kolekcji. Dzięki strumieniom istnieje możliwość korzystania z operacji typu [2] *map-reduce,* służących do przepakowywania i obróbki danych.

Ze względu na fakt wprowadzenia tak dużej liczby zmian w stosunku do poprzedniej wersji *Javy,* nasuwa się pytanie o wydajność. Jak szybko nowe metody radzą sobie z danymi w stosunku do dawniej używanych implementacji.

2. Przegląd literatury

W ostatniej dekadzie przemysł skupił się na produkcji procesorów wielordzeniowych w celu zwiększenia wydajności [2]. Wcześniej próbowano zwiększać szybkość taktowania procesorów, co skutkowało malejącymi zyskami z powodu czynników takich jak: zużycie energii, które wzrasta wykładniczo wraz ze wzrostem częstotliwości. Taki obrót spraw spowodował powstanie równoległych sposobów przetwarzania danych. JDK w wersji 8.0 zostało wydane w marcu 2014. Api *Javy 1.8* wprowadza przejrzyste i bezpieczne wątkowo metody dostępne dla użytkowników nie posiadających zaawansowanej wiedzy w zakresie programowania równoległego [2]. Metody te bazują na stylu programowania funkcyjnego poprzez wykorzystanie operacji typu *map-reduce* [2]. Operacje tego typu istniały już w innych językach programowania, a zaczęły być powszechnie stosowane w miarę wzrostu zapotrzebowania przetwarzania równoległego. Wielu ekspertów twierdzi że wprowadzenie nowych funkcji programowania t.j. wyrażeń *lambda*, operacji *map-reduce* oraz *strumieni,* jest pod wieloma względami, największą zmianą w języku *Java* w ciągu ostatnich kilku lat.

W [3] autorzy zwracają uwagę na olbrzymią rolę jaką odegrał *JIT (just-in-time) compiler* w języku programowania jakim jest *Java.* Dzięki takiemu narzędziu język ten zyskał bardzo na wydajności. W momencie ukazania się *Javy 1.8* autorzy postanowili stworzyć własny kompilator *JIT*

wykorzystujący procesor graficzny (GPU) [3]. Dzięki nowemu API dostarczonemu w *Javie 1.8*, istnieje więcej możliwości programowania równoległego niż w poprzednich wersjach.

W [4] został poruszony temat wydajności *Javy* w odniesieniu do operacji liczbowych. *Java* jest językiem interpretowanym co może powodować powolne operacje na liczbach [4]. Maszyna wirtualna *HotSpot* implementuje nowy i ulepszony *garbage collector* oraz posiada ulepszoną synchronizację. Rozwiązuje także niektóre problemy wydajnościowe za pomocą adaptacyjnej technologii optymalizacji [5]. Większość aplikacji wykonuje tylko niewielką część kodu źródłowego. *HotSpot* posiada interpreter, z którego pomocą aplikacja zostaje uruchomiona. Następnie z jego pomocą kod jest analizowany w kierunku wyszukiwania "gorących punktów". Następnie miejsca te są kompilowane do kodu natywnego. Ponadto *HotSpot* zawiera wszystkie klasyczne optymalizacje, takie jak: eliminacja martwego kodu, wspólna interpretacja, rozwijanie pętli.

W [6] można znaleźć informacje o czynnikach przeszkadzających *Javie* osiągnąć wysoką wydajność. Jednym z nich jest dynamiczne zarządzanie pamięcią. Wszystkie obiekty *Javy* są przydzielone do puli i podlegają grupowaniu, przenoszeniu i odśmiecaniu. Tablice wielowymiarowe są w rzeczywistości wektorami obiektów odwołujących się do obiektów tablicowych o niższym poziomie [6]. Nie ma zatem możliwości zagwarantowania że obiekty te są przyporządkowane do pamięci o takim samym rozmiarze. Tak więc wszelkie algorytmy i transformacje programowo rzadko zależne od wiersza wydają się być przenośne, trudno jest zagwarantować, że takie przekształcenia są w rzeczywistości możliwe i skuteczne. Ponadto mimo gwarancji założenia pewnego układu pamięci, może on zostać unieważniony przez przydziały i odśmiecanie pamięci.

Oryginalnym modelem *Javy* jest interpretacja kodu na kod bajtowy [6]. Kompilatory *JIT* zwykle przekładają kod stosu JVM na kod rejestru RISC dla zoptymalizowania realizacji operacji. Jednak w przeciwieństwie do C, nie jest jasne czy oryginalna optymalizacja na poziomie kodu źródłowego wykonana przez użytkownika czy tłumaczenie wysokiej jakości kodu źródłowego do kodu bajtowego jest skuteczne. Innymi słowy, nie jest jasne, czy jakikolwiek rodzaj optymalizacji na poziomie użytkownika lub na wysokim poziomie spowoduje zwiększenie wydajności, czy utratę wydajności [5].

3. Cel i zakres badań

Celem badań jest porównanie wydajności dwóch wersji *Javy* - 1.7 i 1.8. Ten język programowania w ostatnim czasie rozwija się bardzo dynamicznie. Zmiany pomiędzy wersjami są znaczne i wprowadzają wiele udogodnień lub też tworzą całkowicie nowe możliwości. Pomiędzy w/w wersjami widać różnice, które zmieniają sposób programowania w tym języku, jak również umożliwiają wybór sposobu rozwiązywania problemów.

Zakres pracy obejmuje następujące zagadnienia:

- Analizę wydajności operacji na kolekcjach.
- Porównanie szybkości działania kolekcji z Javy 1.7 i Stream Api z Javy 1.8. [1]
- Wykazanie różnic pomiędzy wersjami Javy.
- Analizę możliwości nowego Api.

Do badania szybkości wykonywania operacji i metod użyto biblioteki *JMH - Java Microbenchmark Harness* [6].

4. Obiekt badań

W celu zbadania wydajności dwóch wersji Javy, zostały stworzone dwie aplikacje zawierające testy porównawcze oparte na bibliotece JMH. *Java Microbenchmark Harness* [6]. Jest to narzędzie, które pomaga poprawnie tworzyć metody do analizy porównawczej. Narzędzie to zostało stworzone przez tą samą grupę osób, która stworzyła wirtualną maszynę Javy.

Napisanie dobrej metody do analizy porównawczej nie jest rzeczą łatwą. Istnieje wiele sposobów optymalizacji, które stosuje wirtualna maszyna lub warstwa sprzętowa, dla komponentów wyizolowanych do celów analizy. Takie optymalizacje mogą nie mieć miejsca w podobnych metodach wykorzystywanych jako część większej aplikacji. Źle stworzona analiza metody może spowodować że jej wynik będzie świadczył o znacznie wyższej wydajności niż jest ona w rzeczywistości. Poprawne pisanie metod do analizy wydajnościowej JVM pociąga za sobą konieczność uniemożliwienia wirtualnej maszynie oraz zasobom sprzętowym optymalizacji, która nie mogłaby być zastosowana w systemie produkcyjnym [7]. Podczas tworzenia metod analizy należy pamiętać że kompilując kod *Javy* do postaci kodu bajtowego, jest on optymalizowany. Metody które przetwarzają dane lecz nie zwracają żadnych wyników wykonanych operacji są automatycznie traktowane przez kompilator jako martwy kod, co sprawia że w rezultacie metody te wydają się szybkie i optymalne. W celu uniknięcia takich błędów należy zadbać o to aby metody analizy zwracały dane. Istnieje również inny sposób wymyślony przez twórców JMH. Na potrzeby kompilacji została utworzona klasa o nazwie [6] *Blackhole*. Jej zadaniem jest konsumowanie powstałych, w procesie analizy, danych. W ten sposób można łatwo uniknąć martwego kodu.

5. Metody testujące wydajność JVM

Aplikacja generująca dane zwraca kolekcję encji o strukturze przedstawionej na listingu 1.

Przykład 1. Testowa encja danych.

```
public class Person {

  private Long personId;
  private String firstName;
  private String lastName;
  private Integer age;
  private Sex sex;
  private Date birthDate;
  private Integer growth;
  private List<Person> children;
  private Person partner;

}
```

Metoda generująca zestaw danych tworzy kolekcję obiektów wypełnioną przypadkowo wygenerowanymi wartościami. Stworzone w ten sposób kolekcje posiadają strukturę umożliwiającą operowanie takimi metodami jak np: przepakowywanie czy wyszukiwanie w różnych warstwach danych.

Analiza wydajności została przeprowadzona na kolekcji zawierającej 100 000 rekordów. Wygenerowanie takiej ilości danych trwa około 30 sekund. JVM potrzebuje czasu aby zacząć działać optymalnie. W związku z tym przeprowadzone testy zostały poprzedzone 5 iteracjami rozgrzewającymi wirtualną maszynę *Javy*, a następnie wykonywano 20 iteracji testowych. Uzyskany w ten sposób rezultat został przedstawiony w mikrosekundach / operację jako średni czas wykonania metody.

5.1. Przepakowanie kolekcji do listy par.

Test przepakowania listy zawierającej encje typu Person został zrealizowany w sposób przedstawiony na listingu 2. Nowo powstała kolekcja składała się z par rekordów.

Przykład 2. Przepakowanie listy do kolekcji par - *Java 1.8.*

```
@BenchmarkMode({Mode.AverageTime})
@OutputTimeUnit(TimeUnit.MICROSECONDS)
@State(Scope.Thread)
public class CollectionsOperations {
  @Benchmark
  public List<Pair<Person, Person>> pairPersons() {
     return Data.getPersons().stream()
           .reduce(new ArrayList<>(), (sum, element) -> {
              sum.add(
               Pair.newPair(element, element.getPartner())
              );
              return sum;
           }, (sum1, sum2) -> {
              sum1.addAll(sum2);
              return sum1;
           });
  }
}
```

Zwrócone przez generator dane traktowane są jako widok. W następnym kroku z pomocą metody *reduce* tworzona jest kolekcja o zadanej strukturze. Kolejne parametry definiują sposób w jaki ma zostać przetworzona lista aby otrzymać spodziewany efekt. Podobnie zostało to zrealizowane w *Javie 1.7.* Listing 3 prezentuje implementację testu.

Przykład 3. Przepakowanie listy do kolekcji par - *Java 1.7.*

```
@BenchmarkMode({Mode.AverageTime})
@OutputTimeUnit(TimeUnit.MICROSECONDS)
@State(Scope.Thread)
public class CollectionsOperations {

  @Benchmark
  public List<Pair<Person, Person>> pairPersons() {
     final List<Pair<Person, Person>> personPairs =
                                        new ArrayList<>();
     for (final Person person : Data.getPersons()) {
        personPairs.add(Pair.newPair(person,
person.getPartner()));
     }
     return personPairs;
  }
}
```

5.2. Zmiana wartości logicznej.

Zmiana wartości zmiennej logicznej na przeciwną została zaimplementowana jako jeden z testów analizy wydajnościowej. Sposób w jaki został stworzony ten test został przedstawiony na listingu 4.

Przykład 4. Zmiana wartości logicznej - *Java 1.8 i 1.7.*

```
@State(Scope.Group)
@BenchmarkMode({Mode.AverageTime})
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@Threads(2)
public class PingPong {
  private static final String pingPong = "pingPong";
  private final AtomicBoolean boolFlag = new
AtomicBoolean();

  private static void doNothing() {
  }

  @Benchmark
  @Group(pingPong)
  public void ping(Control cnt) {
     while (!cnt.stopMeasurement &&
           !boolFlag.compareAndSet(false, true)) {
        doNothing();
     }
  }
  @Benchmark
  @Group(pingPong)
  public void pong(Control cnt) {
     while (!cnt.stopMeasurement &&
           !boolFlag.compareAndSet(true, false)) {
        doNothing();
     }
  }
}
```

W tym teście dwa wątki zmieniają wartość logiczną zmiennej gdy spełniony jest prosty warunek. Do poprawnego działania testu niezbędne jest skorzystanie z bezpiecznej wątkowo implementacji *AtomicBoolean.* Dodatkowo dwie metody posiadają tą samą grupę dzięki czemu jedna iteracja wywołuje obie metody jeden raz. Jedynym parametrem obu tych funkcji jest klasa *Control*, która przechowuje 2 flagi - *startMeasurement* oraz *stopMeasurement* [6]. Dzięki niej można zatrzymać wykonywanie testu unikając nieskończonej pętli. Powyższy test sprawdza szybkość zmiany wartości logicznej pomiędzy dwoma wątkami.

5.3. Zmiana wartości liczbowej.

Następny test sprawdza szybkość zmiany wartości liczbowej z 0 na 1 i odwrotnie. Test wygląda analogicznie jak poprzedni, który sprawdzał szybkość zmiany wartości logicznej. Kod odpowiadający za analizę przykładu znajduje się na listingu 5.

Przykład 5. Zmiana wartości liczbowej - *Java 1.8 i 1.7.*

```
@State(Scope.Group)
@BenchmarkMode({Mode.AverageTime})
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@Threads(2)
public class PingPong {
```

```
  private static final String intPingPong = "intPingPong";

  private final AtomicInteger intFlag = new
AtomicInteger();

  private static void doNothing() {

  }

  @Benchmark
  @Group(intPingPong)
  public void intPing(Control cnt) {
    while (!cnt.stopMeasurement &&
!intFlag.compareAndSet(0, 1)) {
      doNothing();
    }
  }

  @Benchmark
  @Group(intPingPong)
  public void intPong(Control cnt) {
    while (!cnt.stopMeasurement &&
!intFlag.compareAndSet(1, 0)) {
      doNothing();
    }
  }
}
```

Podobnie jak poprzednio, dwa wątki konkurujące ze sobą zmieniają wartość liczbową, co zostało przedstawione na powyższym przykładzie. Jednokrotne wywołanie testu powoduje uruchomienie pięciu iteracji rozgrzewających wirtualną maszynę *Javy,* a następnie dziesięciu iteracji testowych, z których każda uruchamia jeden raz metodę *intPing* i *intPong*. Wynikiem testu jest średni czas wykonania operacji mierzony w nanosekundach. Test ten ma na celu zmierzenie czasu potrzebnego na zmianę wartości zmiennej liczbowej.

6. Wyniki badań

Badania zostały przeprowadzone z wykorzystaniem dwóch wersji *Javy*. Pierwszą z nich była wersja JDK 1.7.0_80, VM 24.80-b11, a drugą JDK 1.8.0_131, VM 25.131-b11. Ponadto testy zostały przeprowadzone na dwóch różnych stacjach roboczych - komputerze stacjonarnym oraz laptopie. Komputer stacjonarny posiadał system operacyjny *Windows 7 Home Premium 64 bit*, procesor *Intel® Core™ i5-4460 CPU @3.20 GHz 3.20 GHz,* 16,0 GB pamięci RAM, a laptop - system operacyjny *Ubuntu 16.04 LTS 64 bit,* procesor *Intel® Core™ i7-2860QM CPU @2.50 GHz x 8, 32,0 GB pamięci RAM.*

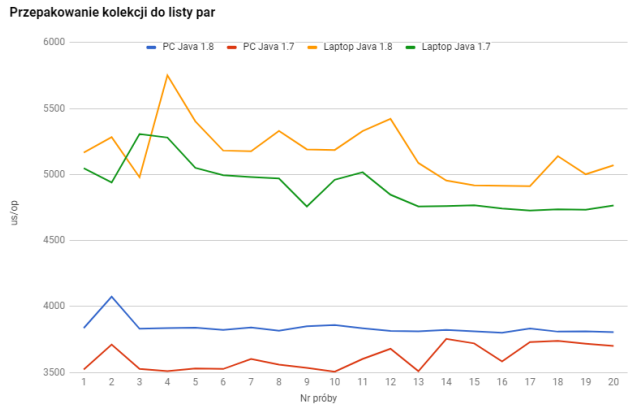
Każdy test składał się z 5 iteracji rozgrzewających wirtualną maszynę *Javy*, 20 iteracji pomiarowych i został wykonany jeden raz. Testy były wykonywane z wykorzystaniem jednego wątku, z wyjątkiem tych metod, które potrzebowały więcej niż jednego wątku, co zostało jednoznacznie zaznaczone przy opisie metody.

6.1. Przepakowanie kolekcji do listy par.

Operacja przepakowania kolekcji zawierającej encje *Person* do listy par dwóch encji *Person* w sposób pokazany na listingu 2, dostarczyła wyniki, które zostały zaprezentowane w tabeli 1. Rys. 1 obrazuje dane z tabeli. Na tej podstawie można wywnioskować że najszybciej test został wykonany na maszynie wirtualnej *Javy 1.7* na komputerze stacjonarnym. Najwięcej czasu potrzebowała *Java 1.8* na laptopie. Można także zaobserwować mniejszą wydajność laptopa w odniesieniu do komputera stacjonarnego. Odchylenia standardowe wyników testu wahają się od 80,00 do 200,00.

Tabela 1. Wyniki operacji przepakowania kolekcji do listy par.

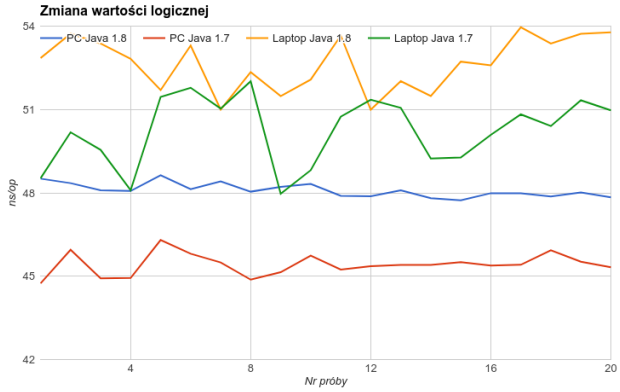
Nr próby	PC Java 1.8	PC Java 1.7	Laptop Java 1.8	Laptop Java 1.7
1	3836,142 us/op	3522,768 us/op	5165,445 us/op	5047,974 us/op
2	4073,756 us/op	3711,410 us/op	5282,647 us/op	4939,661 us/op
3	3830,732 us/op	3526,263 us/op	4979,250 us/op	5307,038 us/op
4	3835,686 us/op	3510,052 us/op	5751,930 us/op	5280,036 us/op
5	3838,824 us/op	3530,200 us/op	5403,282 us/op	5050,131 us/op
6	3822,842 us/op	3526,287 us/op	5181,453 us/op	4994,288 us/op
7	3841,086 us/op	3601,329 us/op	5176,269 us/op	4980,575 us/op
8	3816,005 us/op	3559,296 us/op	5330,672 us/op	4970,284 us/op
9	3849,990 us/op	3534,492 us/op	5190,490 us/op	4757,268 us/op
10	3858,492 us/op	3505,599 us/op	5185,868 us/op	4960,080 us/op
11	3835,131 us/op	3602,111 us/op	5329,542 us/op	5016,783 us/op
12	3814,478 us/op	3679,456 us/op	5421,572 us/op	4846,614 us/op
13	3812,144 us/op	3509,353 us/op	5087,040 us/op	4757,908 us/op
14	3822,110 us/op	3754,258 us/op	4954,094 us/op	4760,562 us/op
15	3811,074 us/op	3719,913 us/op	4918,205 us/op	4766,285 us/op
16	3801,343 us/op	3582,895 us/op	4915,186 us/op	4741,753 us/op
17	3832,891 us/op	3730,054 us/op	4912,205 us/op	4727,307 us/op
18	3809,141 us/op	3738,729 us/op	5138,977 us/op	4736,572 us/op
19	3810,941 us/op	3717,606 us/op	5001,970 us/op	4732,542 us/op
20	3804,932 us/op	3699,994 us/op	5070,013 us/op	4765,845 us/op
średnia	3837,887 us/op	3613,103 us/op	5169,805 us/op	4906,975 us/op



Rys 1: Wyniki operacji przepakowania kolekcji do listy par.

6.2. Zmiana wartości logicznej.

Wyniki testu zmieniającego wartość logiczną zmiennej przez dwa wątki zostały pokazane na rys. 2. Należy zwrócić uwagę na jednostkę w jakiej zostały przedstawione wyniki. Dotychczas były to µs/op, natomiast w tym przypadku są to ns/op, ze względu na bardzo krótki czas wykonania jednej operacji. Na wykresie widać wyraźnie niewielką przewagę

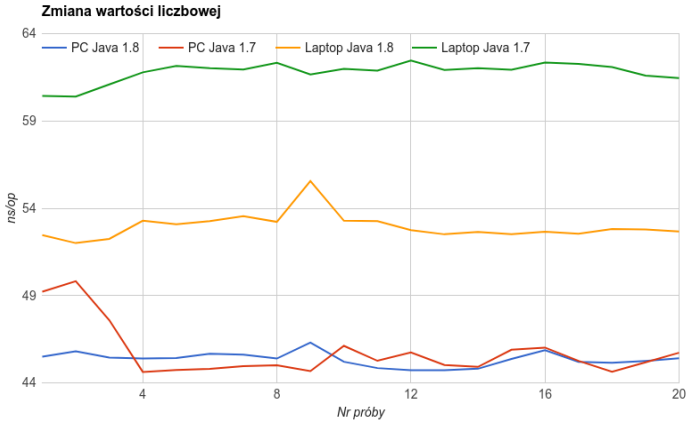


Rys 2: Zmiana wartości logicznej

Javy 1.7 nad *Javą 1.8,* oraz przewagę komputera stacjonarnego nad laptopem. Wartości odchylenia standardowego dla poniższych wyników to ok. 0,3 dla PC i ok 2,0 dla laptopa. Świadczy to o mniejszych wahaniach wyników na komputerze stacjonarnym.

6.3. Zmiana wartości liczbowej.

Rezultaty uzyskane podczas wykonywania testu zmieniającego wartość liczbową zmiennej z 0 na 1 przez dwa wątki, zostały zaprezentowane na rys.3. Na podstawie wykresu można zauważyć, że najwięcej czasu potrzebowała *Java 1.7* na laptopie. *Java 1.7 i 1.8* na PC osiągnęły podobne rezultaty. Wszystkie odchylenia standardowe dla wyników przeprowadzonych testów oscylowały wokół wartości 1,00, co świadczy o niewielkim odchyleniu wyników od średniej.



Rys 3: Zmiana wartości liczbowej

7. Wnioski

Celem testów było zbadanie wydajności wirtualnej maszyny Javy w zależności od użytej wersji oraz platformy uruchomieniowej. Do testów zostały użyte dwie wersje Javy - 1.7 i 1.8, a testy były uruchamiane na dwóch różnych komputerach - komputerze stacjonarnym i laptopie. Na potrzeby testów zostały stworzone trzy niewielkie projekty. Pierwszy projekt został stworzony z myślą o dostarczeniu danych do testów. Kolejne dwa projekty zawierały metody testowe oparte o *Java Microbenchmark Harness* - framework służący do próbkowania i badania wydajności metod i klas *Javy* [7], oraz różne wersje *JDK*.

Na podstawie otrzymanych rezultatów można zaobserwować wyższą wydajności w *Javie 1.8* podczas wykonywania metod służących do przepakowania danych z jednych struktur do innych. Jednocześnie zauważono że wydajność ta jest mniejsza niż w Javie 1.7, w testach zmieniających wartość zmiennych logicznych. Otrzymane wyniki testów posiadały zazwyczaj niewielkie wartości odchylenia standardowego co świadczy o niewielkiej amplitudzie wartości. Wszystkie testy zostały wykonane jeden raz. Każdy test posiadał także podsumowanie zawierające najmniejszy, średni i największy czas wykonania oraz uzyskane odchylenie standardowe. W większości testów, do ich wykonania, użyty został jeden wątek a wyniki zostały przedstawione w jednostce µs/op. Testy zmiany wartości zmiennych liczbowych i logicznych, wykorzystywały dwa lub więcej wątków oraz ich wyniki przedstawione zostały w jednostce ns/op ze względu na bardzo krótki czas wykonania.

Utworzone projekty z testami wydajnościowymi można dowolnie rozwijać implementując kolejne testy sprawdzające wydajność wirtualnej maszyny *Javy*. Na potrzeby artykułu powstały szkielety aplikacji, gotowe do dalszego rozwoju. Tworzenie testów wydaje się dosyć łatwe pod warunkiem zachowania zasad poprawności ich implementowania. Należy pamiętać m.in. aby metody testowe były jak najbardziej zbliżone do produkcyjnie wykorzystywanych metod. Kryteria którymi należy się kierować to przede wszystkim konieczność zwracania danych [6], lub korzystanie z możliwości jakie dostarcza *JMH,* czyli *blackhole,* które konsumują wyniki.

W przypadku niezastosowania powyższych zasad, powstały kod jest traktowany jako *martwy* [6], co powoduje błędne wyniki testów, ze względu na optymalizację zastosowaną przez *JVM*. Duży wpływ na otrzymane rezultaty ma też środowisko na jakim testy zostaną uruchomione. Mocniejsze maszyny osiągają znacznie lepsze rezultaty niż komputery o słabszej wydajności i specyfikacji sprzętowej.

Istnieje możliwość poprawienia wydajności wirtualnej maszyny poprzez dostosowanie opcji uruchomieniowych. Dzięki temu widać możliwość rozwinięcia pracy o próby konfiguracji *JVM* tak, aby osiągnąć lepsze rezultaty powstałych testów. Takie opcje można zmieniać w trakcie wykonywania testów, co pozwala na dynamiczne dostosowywanie wirtualnej maszyny, w sposób pozwalający na optymalizację wydajności. Przykładowo można stworzyć metodę opartą na algorytmie genetycznym, wybierającą najlepsze ustawienia i modyfikującą je w niewielki sposób, w poszukiwaniu wydajniejszych ustawień.

Literatura

[1] Java Platform, Standard Edition (Java SE) 8, 1993, 2016, Oracle and/or its affiliates, http://docs.oracle.com/javase/8/docs/api/

[2] Andres R. Masegosa, Ana M. Martinez, Hanen Borchani, Probabilistic Graphical Models on Multi-Core CPUs Using Java 8, 2016, IEEE

[3] Kazuaki Ishizaki, Akihiro Hayashi, Gita Koblents, Vivek Sarkar, Compiling and Optimizing Java 8 Programs for GPU Execution, 2015, IEEE

[4] Bogdan Oancea, Ion Gh. Rosca, Tudorel Andrei, Andreea Iluzia Iacob, Evaluating Java performance for linear algebra numerical computations, 2011, ScienceDirect

[5] Kuo-Yi Chen, J. Morris Chang, Ting-Wei Hou, Multithreading in Java: Performance and Scalability on Multicore Systems, 2011, IEEE

[6] "JMH", 2017, Oracle Corporation and/or its affiliates, http://openjdk.java.net/projects/code-tools/jmh/

[7] "JMH - Java Microbenchmark Harness", 2015, Jakob Jenkov, http://tutorials.jenkov.com/java-performance/jmh.html

[8] Guillermo L. Taboada, Sabela Ramos, Roberto R. Expósito, Juan Touriño, Ramón Doallo, Java in the High Performance Computing arena: Research, practice and experience, 2013, ScienceDirect

[9] Satoshi Matsuoka, Shigeo Itou, Towards performance evaluation of high-performance computing on multiple Java platforms, 2001, ScienceDirect

[10] "Collections", 2014, Oracle, Inc, http://docs.oracle.com/javase/tutorial/collections/intro/index.html

[11] "Paralellism", 2014, Oracle,Inc, http://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html

[12] "Reduction", 2014, Oracle, Inc, http://docs.oracle.com/javase/tutorial/collections/streams/reduction.html