FRI-ONLINE-1-CCT1-12

INVESTIGATION OF THE EFFICIENCY OF DIFFERENT METHODS FOR DATA STORAGE AND DATA PROCESSING IN THE JAVA PROGRAMMING LANGUAGE[12]

Assoc. Prof. Georgi Georgiev, PhD
Dept. of Computer Systems and Technologies,
University of Ruse
Tel.: 082-888 744
E-mail: gtgeorgiev@ecs.uni-ruse.bg

Abstract: *The paper investigates the efficiency in terms of speed and memory usage of various methods for storing large amounts of data in Java - arrays and different types of collections. Lambda expressions are an exciting new feature, introduced in Java 8, but there aren't many conclusive proofs whether there is any significant performance gain from using them. This paper attempts to provide an unbiased comparative analysis of the efficiency in terms of speed of data processing by different means - classical and lambda expressions. Tests have been performed on several computer systems providing different processing power in an attempt to eliminate the effect of the unerlying platform on code efficiency. Conclusions are made regarding the most efficient means for data storage and data processing in Java, with an accent on Lambda expressions.*

Keywords: *Efficiency, Data Storage, Data Processing, Lambda Expressions, Java*

JEL Codes: *C80*

1. ВЪВЕДЕНИЕ

Една гледна точка към ламбда изразите (ЛИ, lambda expressions) е, че те представляват анонимни функции, които могат да се предават като параметри на други функции. Концепцията е присъща на езиците за функционално програмиране като LISP (създаден през 1958 г.) и от по-съвременните, haskell. Функциите като параметри (без анонимността обаче) съществуват в езика C от създаването му през 1978 г. благодарение на функционалния тип данни.

В програмния език Java ЛИ се появяват чак през г. във версия 8 на езика през март 2014 г. За много от програмистите, използващи Java и особено за по-младите от тях, ЛИ са вълнуваща новост. Разработчиците от Oracle също отделят доста внимание на удобствата, които ЛИ предоставят в различни страници на уебсайта си, както самоучители така и маркетингово ориентирани.

По мнение на автора, за въвеждане на ЛИ в Java без нарушаване на съвместимостта с предишни версии, са жертвани някои основни принципни положения, а именно изчистената реализация на обектно-ориентирания модел, която Java предлагаше във версиите до 7.

2. ЦЕЛ И ЗАДАЧИ НА ИЗСЛЕДВАНЕТО

Целта на настоящото изследване е да се провери дали ЛИ дават някакво предимство по отношение на производителност, т.е. скорост на изпълнение на програмен код, обработващ различни обеми от данни. Подобни цели си поставят и *Ward et al. 2015*, и *Jurinová, 2018*. Още едно напълно независимо изследване на този въпрос би могло да послужи на програмисти или преподаватели по програмиране.

За постигане на целта трябваше да се решат няколко задачи:

[12] Докладът е представен на 13 ноември 2020 с оригинално заглавие: INVESTIGATION OF THE EFFICIENCY OF DIFFERENT METHODS FOR DATA STORAGE AND DATA PROCESSING IN THE JAVA PROGRAMMING LANGUAGE

- Да се подбере подходящ пример за обработка на данни.
- Да се избере някоя от конкретните реализации на колекции в Java, в която да бъдат записани данните.
- Да се състави програма на Java, която да тества различни методи на обработка на данните в колекцията.
- Да се проведат тестове за бързодействие с различни по обем данни и на различни по производителност компютърни системи. Тези тестове да се използват и за приблизителна оценка на заеманата от данните памет при съхранението им в списък (какъвто по същество е `ArrayList`) и в масив.
- Да се верифицират резултатите, най-малкото като се провери повторяемостта им.
- Да се обработят получените резултати и да се направят изводи.

3. ИЗЛОЖЕНИЕ

3.1. Изходни данни, представяне и обработка

Като изходни данни беше избрано графично изображение, а като обработка - прилагане на филтър към него. Избран беше най-елементарният филтър - преобразуване на цветно изображение до изображение в нива на сивото. Така се получават следните предимства:

- Обработката на информация е част от реална функционалност, предлагана от стандартни приложни програми.
- Броят обработвани елементи може да варира в изключително широки граници.
- Дава се възможност за нагледно и бързо верифициране на резултата.
- Същинската обработка на информация е елементарна и включва: 2 сумирания, 1 деление и 6 операции за достъп до паметта

Във вид на код изглежда така:

```
r = g = b = ( r + g + b ) / 3;
```

Аритметичните операции би трябвало даже и в Java да се изпълняват за един процесорен такт. Така времето за обработка всъщност се свежда основно до достъпа до елемента (пиксела) от изображението в рамките на избраната структура от данни и активирането на метода, извършващ обработката.

За представяне на данните (пикселите от изображението) е използван `java.util.ArrayList` като най-елементарната и най-често използвана конкретна реализация на колекция в Java. Направен е за сравнение и тест, при който данните се съхраняват в класически масив.

Изследвани са 3 графични изображения в JPEG формат. Изходният формат на изображенията всъщност е без значение, тъй като обработката се извършва след декодиране до bitmap. Параметрите на изображенията са дадени в Таблица 1.

Таблица 1 - Използвани изображения

Файл	Резолюция	Размер в байтове като bitmap	Размер на файла
img1.jpg	1 920 x 1 200	27 648 000	350kB
img2.jpg	3 264 x 2 448	95 883 264	1 680kB
img3.jpg	12 160 x 6 080	887 193 600	11 000kB

При пресмятане на размера в байтове се има предвид 24 битов цвят в RGB формат и заделено по едно цяло число (`int`, 4 байта) за всяка компонента от цвета.

3.2. Тествани системи

Беше преценено, че може да представлява интерес изследването на производителността да бъде направено върху коренно различни по производителност и предназначение системи.

Тествани са 4 системи: сравнително мощен десктоп компютър, два лаптопа с много близки параметри и процесори, акцентиращи на енергийна ефективност за сметка на производителност, както и нетбук система. Спецификациите им са в Таблица 2.

Таблица 2 - Тествани компютърни конфигурации

№	Фамилия и модел на процесора	Тактова честота	Брой ядра / нишки	RAM	Диск
1	i5-8400	2,8÷4,0GHz	6C/6T	16GB 3333MHz	SSD
2	i5-4200U	1,6÷2,3GHz	2C/4T	4GB 1600MHz	SSD
3	Pentium 3825U	1,9GHz	2C/4T	8GB 1600MHz	SSD
4	Atom x5-Z8300	1,44÷1,84GHz	4C/4T	2GB 1600MHz	SSD

3.3. Метод на тестване

- Всички системи са рестартирани преди началото на серията тестове, но не и след всеки отделен тест.
- Във всички случаи операционната система е 64 битов Windows 10 с минимални вариации в подверсията.
- От настройките за управление на консумацията е зададена "Максимална производителност", с изключение на най-бавната система, където такава опция липсва.
- Изключени са както безжичната карта, така и Ethernet контролера, за да няма натоварване на системите заради обмен по мрежата, например изтегляне на обновявания на операционната система.
- Всички тестове са направени от конзолен прозорец при нито един стартиран друг процес.
- Използвана е 64 битова версия на Java 8, по-точно JRE 1.8.0_121-b13.

3.4. Изпълнени тестове

Над всяко от трите изображения и на всяка от четирите компютърни системи са изпълнени по 6 теста, представени в Таблица 3. Тествани са 5 различни начини за обработка на колекции, а за теста с масива, разбира се, е използвана класическа обработка с `for` цикъл.

Таблица 3 - Проведени тестове

№	Описание	Код

1	`ArrayList` с класически `for` цикъл	`for (int i=0; i<imageAL.size(); i++){`
		`  Pixel pxl = imageAL.get(i);`
		`  pxl.toGray();`
		`}`

2	`ArrayList` с "нов" (enhanced) `for` цикъл	`for (Pixel pxl : imageAL){`
		`  pxl.toGray();`
		`}`

3	`ArrayList` с итератор	`Iterator<Pixel> iter=imageAL.iterator();`
		`while (iter.hasNext()){`
		`  iter.next().toGray();`
		`}`

4	ArrayList с forEach и обект от Consumer	`imageAL.forEach(new Consumer<Pixel>() { public void accept(Pixel pxl){ pxl.toGray(); } });`
5	ArrayList с forEach и lambda expression	`imageAL.forEach((e) -> { e.toGray(); });`
6	Масив с класически for цикъл	`for (int i = 0; i < imageARR.length; i++) { imageARR[i].toGray(); }`

В кода може да се забележи клас `Pixel`. Това е собствен клас - минимална обвивка на един пиксел от изображението с дефиниран метод `toGray()`, който извършва елементарната аритметика по преобразуването на пиксела до нюанс на сивото.

Графичното изображение се прочита от диска и се прехвърля в `ArrayList`-а или масива преди всеки отделен тест, а след обработката му се освобождава заеманата памет, включително чрез пряко активиране на механизма за управление на динамичната памет (Garbage Collector).

Времената за прочитане на изображението от диска, показването му на екрана и прехвърлянето му в `ArrayList` или масив, разбира се, не са включени в изследването. Разгледано е само чистото време за обработка на пикселите.

3.5. Верификация на резултатите

В течение на изследването се очерта поне един проблем, който водеше до принципно различни резултати при всяко следващо изпълнение на тестовете. Това наложи неколкократни сериозни промени по програмния код и начина на провеждане на тестовете.

На системите с под 4GB оперативна памет се забеляза интензивна работа на операционната система с дисковата виртуална памет (swap файла). Проблемът се изявяваше само при обработката на най-голямото изображение. Полезната информация в изображението е около 900MB съгласно Таблица 1. Собствените измервания на заеманата в Java памет очаквано показват още по-голям разход на памет - изображението заема около 2,4GB в `ArrayList` и около 2GB в масив. Проблемът не е предмет на това изследване; допуска се, че е свързан с вътрешната реализация на използвания клас `java.awt.image.BufferedImage` и може би с управлението на паметта от страна на Java интерпретатора (JVM, java.exe) за Windows. За преодоляването му бяха взети следните мерки:

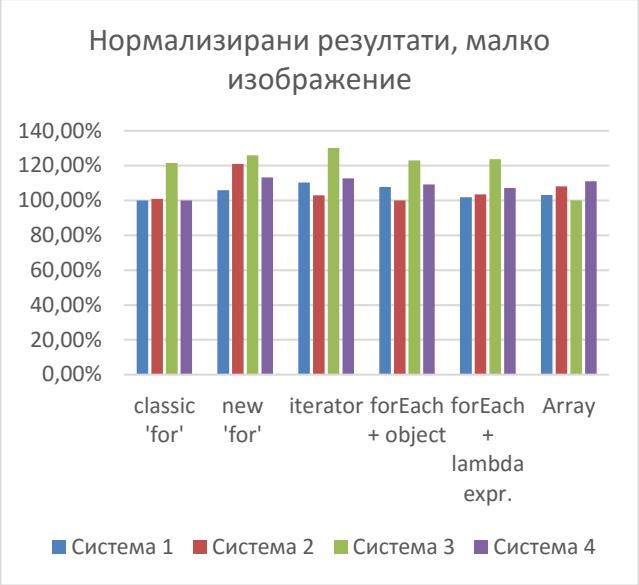
- Беше забелязано, че първото от поредицата от десет изпълнения на всеки от шестте теста завършва за много по-дълго време от следващите - до десет *пъти* по-бавно. Затова при отчитане на резултатите беше взета средно аритметичната стойност не от всичките десет теста, а от осем - с игнориране на най-бавния и най-бързия тест.
- За да работи въобще програмата, се наложи при стартирането на JVM да се изисква размер на динамичната памет от 4GB (java.exe -Xmx4g).

РЕЗУЛТАТИ

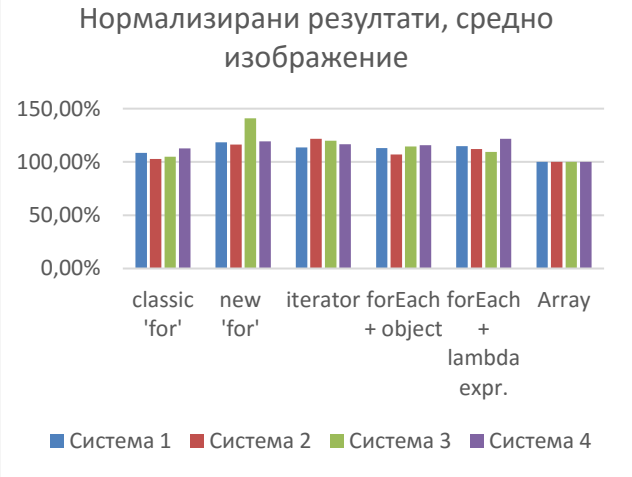
Пълните резултати са представени на Фигура 1, а на Фигури 2, 3 и 4 са дадени нормализираните резултати за трите изображения. За база (100%) е прието най-бързото изпълнение в серията.

			classic 'for'	new 'for'	iterator	forEach + object	forEach + lambda expr.	Array
Система 1 Coffee Lake	img1	от 10 опита	8,24	8,68	9,01	8,69	8,30	8,24
		филтриран (8 опита)	7,81	8,27	8,61	8,41	7,86	8,05
		нормализиран	100,00%	105,89%	110,24%	107,68%	101,92%	103,07%
		заемана памет, MB	89,20	89,17	89,13	89,22	89,09	105,00
	img2	от 10 опита	25,18	27,42	26,33	26,07	26,53	23,04
		филтриран (8 опита)	24,74	26,99	25,94	25,76	26,20	22,82
		нормализиран	108,41%	118,27%	113,67%	112,88%	114,81%	100,00%
		заемана памет, MB	351,29	350,34	351,29	351,29	351,29	291,79
	img3	от 10 опита	199,88	194,93	210,89	214,05	207,35	166,99
		филтриран (8 опита)	198,54	194,32	210,78	213,04	206,17	166,68
		нормализиран	119,11%	116,58%	126,46%	127,81%	123,69%	100,00%
		заемана памет, MB	2170,24	1908,82	2272,56	2166,52	2258,97	1825,81
Система 2 Haswell	img1	от 10 опита	16,04	20,05	16,29	15,65	16,18	16,70
		филтриран (8 опита)	15,38	18,42	15,67	15,23	15,76	16,66
		нормализиран	100,98%	120,95%	102,89%	100,00%	103,48%	108,08%
		заемана памет, MB	53,39	106,60	53,86	53,19	53,39	53,98
	img2	от 10 опита	38,81	43,39	46,63	39,91	41,63	36,77
		филтриран (8 опита)	37,70	42,65	44,67	39,22	41,18	36,69
		нормализиран	102,75%	116,24%	121,75%	106,90%	112,24%	100,00%
		заемана памет, MB	202,90	205,27	217,77	216,73	197,91	194,68
	img3	от 10 опита	1070,00	603,07	550,73	422,70	446,28	422,05
		филтриран (8 опита)	649,68	402,91	357,38	345,01	327,41	417,21
		нормализиран	198,43%	123,06%	109,15%	105,38%	100,00%	127,43%
		заемана памет, MB	2152,56	2117,12	2108,13	2114,22	2101,76	1740,82
Система 3 Pentium	img1	от 10 опита	18,11	18,62	19,28	17,82	18,00	14,35
		филтриран (8 опита)	16,89	17,51	18,10	17,10	17,19	13,90
		нормализиран	121,51%	125,97%	130,22%	123,02%	123,67%	100,00%
		заемана памет, MB	67,16	66,08	67,17	67,17	67,16	46,96
	img2	от 10 опита	48,87	65,15	55,87	53,06	50,99	46,17
		филтриран (8 опита)	47,97	64,53	54,84	52,41	50,12	45,78
		нормализиран	104,78%	140,96%	119,79%	114,48%	109,48%	100,00%
		заемана памет, MB	227,42	228,41	213,95	228,67	193,46	244,59
	img3	от 10 опита	383,03	394,79	398,80	384,60	383,53	340,87
		филтриран (8 опита)	372,56	394,04	397,74	384,20	382,85	340,24
		нормализиран	109,50%	115,81%	116,90%	112,92%	112,52%	100,00%
		заемана памет, MB	2155,35	1921,81	1916,56	1908,76	1921,07	1762,19
Система 4 Atom	img1	от 10 опита	47,11	51,80	52,25	49,59	49,14	49,92
		филтриран (8 опита)	44,80	50,76	50,47	48,95	47,99	49,70
		нормализиран	100,00%	113,30%	112,66%	109,26%	107,12%	110,94%
		заемана памет, MB	71,51	69,11	69,13	67,03	63,71	63,66
	img2	от 10 опита	125,78	132,44	129,69	129,99	135,11	110,07
		филтриран (8 опита)	123,24	130,47	127,41	126,43	133,28	109,41
		нормализиран	112,64%	119,25%	116,45%	115,56%	121,82%	100,00%
		заемана памет, MB	182,47	203,76	183,67	183,47	204,68	184,55
	img3	от 10 опита	33070,00	38340,00	33850,00	34720,00	34040,00	35850,00
		филтриран (8 опита)	32140,00	38080,00	32680,00	33340,00	32150,00	35270,00
		нормализиран	100,00%	118,48%	101,68%	103,73%	100,03%	109,74%
		заемана памет, MB	1889,66	2148,87	1871,33	2162,64	2155,14	1786,57

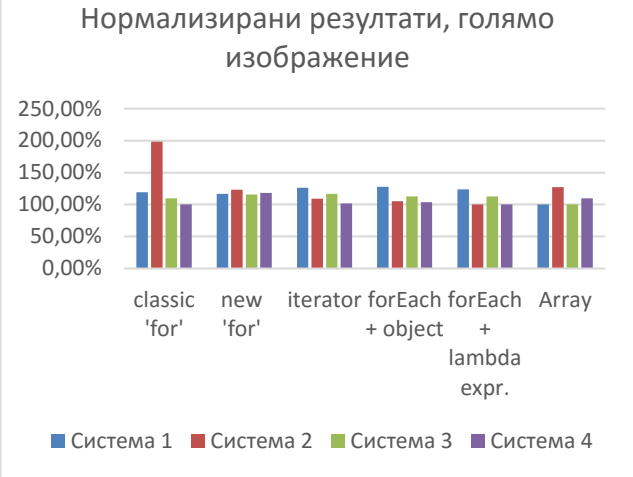
Фигура 1 - Пълни резултати от изследването



Фигура 2 - Нормализирани резултати 1

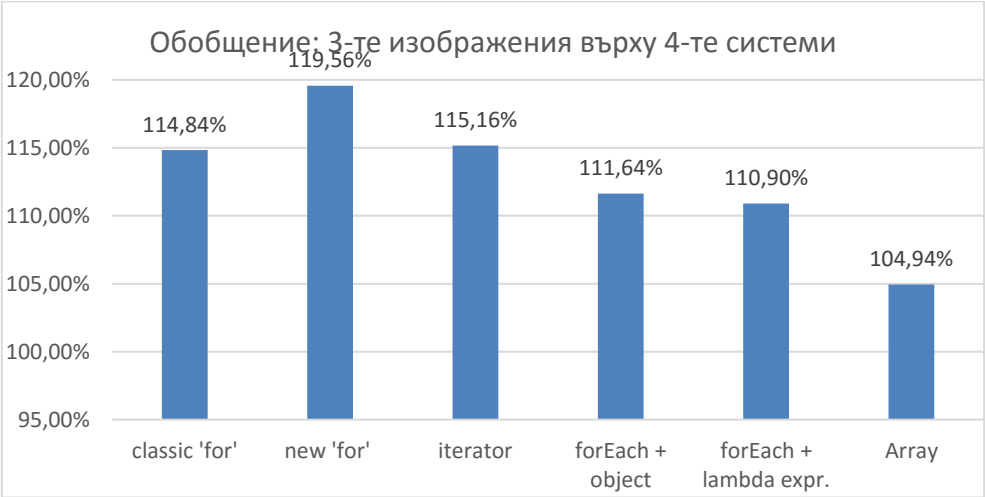


Фигура 3 - Нормализирани резултати 2



Фигура 4 - Нормализирани резултати 3

На Фигура 5 са представени обощените резултати за трите изображения върху четирите тестови компютърни системи.



Фигура 5 - Обобщени резултати за различните методи на обработка на данните

ИЗВОДИ

1. Резултатите в рамките на едно и също изображение и една и съща компютърна конфигурация са прекалено близки, за да могат да се направят категорични изводи относно ефективността на различните методи на достъп до данните.

Все пак, особено при големи обеми от данни, най-ефективна е обработката в масива, следвана от тази в `ArrayList` с ламбда израз.

2. По отношение на заеманата памет, както може и да се очаква, най-оптимален е масивът, макар и не с много. Това обаче се отнася за случаи като настоящия тест, при които броят на елементите е предварително и безусловно известен.

3. Дали кодът става "по-красив" от използването на ЛИ - всеки може да прецени за себе си от Таблица 3. Е, има и случаи, в които отговорът е по-ясен:

```
btn1.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Hello world");
    }
});
```

срещу

```
btn2.addActionListener(e -> System.out.println("Hello lambda"));
```

4. Дали кодът става по-четлив и разбираем от използването на ЛИ - категорично **не** на първо време; след придобиване на известен опит - може би.

5. Добре е, че в процедурните езици намират място концепции от непроцедурното функционално програмиране. Потвърждение на това е фактът, че ЛИ бяха въведени и стандартизирани и в други "класически" езици, например C++ 11 през 2011, три години преди въвеждането им в Java..

6. Настоящото изследване в голяма степен потвърждава изводите, направени от (*Ward et al. 2015)*, и (*Jurinová, 2018)*

ACKNOWLEDGEMENT

This work/paper is supported by project 20-FEEA-01 “Methods and tools for multimedia content analysis, and automated document and big data processing”, funded by the Research Fund of the “Angel Kanchev” University of Ruse.

Този доклад се публикува с подкрепата на проект 20-ФЕЕА-01 „Методи и средства за търсене по мултимедийно съдържание, анализ и автоматизирана обработка на документи и големи масиви от данни“, финансиран от фонд „Научни изследвания“ на Русенски университет „Ангел Кънчев“.

REFERENCES

Ward, A. and D. Deugo. (2015). Performance of Lambda Expressions in Java 8. *Int'l Conf. Software Eng. Research and Practice | SERP'15*.

Jurinová, J. (2018). Performance improvement of using lambda expressions with new features of Java 8 vs. other possible variants of iterating over ArrayList in Java. *Journal of Applied Mathematics, Statistics and Informatics | Volume 14: Issue 1*

Java documentation and Lambda Expression tutorials at Oracle, https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html and others