National and Kapodistrian
UNIVERSITY OF ATHENS

Hw2

Advanced Programming Techniques

Aristotelis Pozidis
AM: CS1200002

December 14th 2020

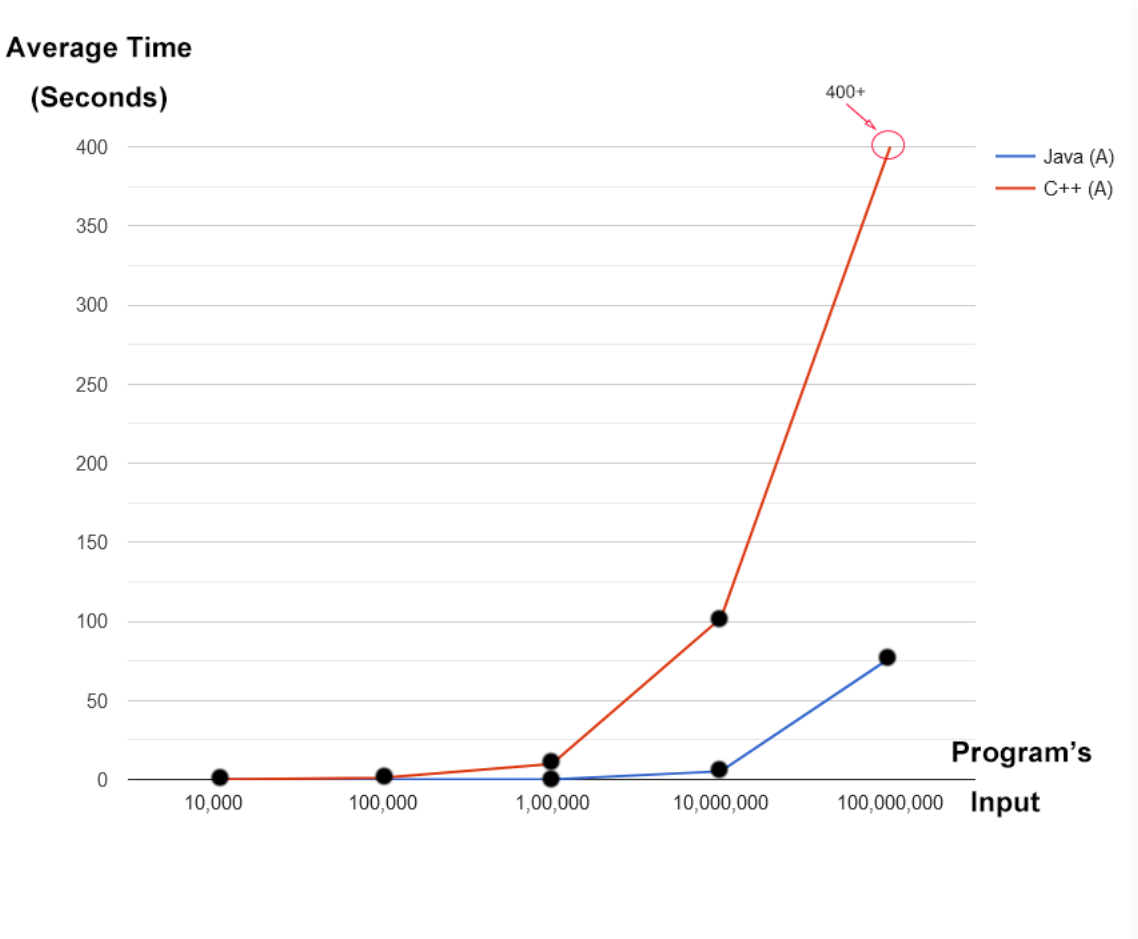
Java vs C++

Program A

The A.java and A.cpp programs are doing something very simple, they create 100 dead object between every alive one. Then they shuffle the array by selecting a random number from 0 - array's size, and replacing the current index with the random index, that way the array won't have allocated the objects in order. Finally they make some calculation with the fields of the object and "kill" the alive object.

Below are indicated the metrics for both programs and the performance of the A.java program with the use of different garbage collectors.

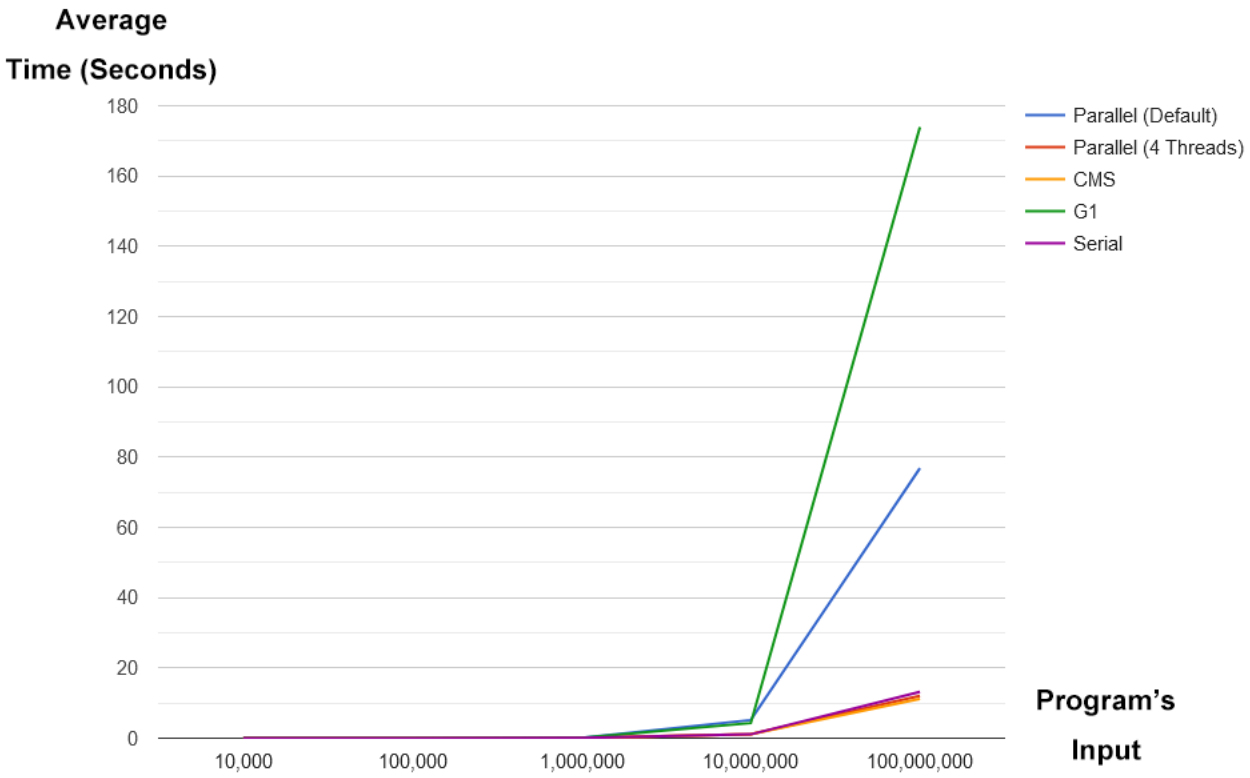
	INPUT	Java (A)	C++ (A)
1	10,000	0	0
2	100,000	0.01	0.98
3	1,00,000	0.054	9.71
4	10,000,000	5.16	101.99
5	100,000,000	76.84	400+



The metrics obviously show that as the program's input is getting increased the C++ performance is getting decreased. On the other hand Java keeps a steady completion time.

Below are the metrics of the Java program with the use of different garbage collectors.

	INPUT	Parallel (Default)	Parallel (4 Threads)	CMS	G1	Serial
1	10,000	0	0	0	0	0
2	100,000	0.01	0.01	0.015	0.014	0.01
3	1,000,000	0.054	0.056	0.055	0.086	0.055
4	10,00,000	5.16	1.147	1.13	4.308	1.1
5	100,000,000	76.84	12.07	11.224	173.89	13.19

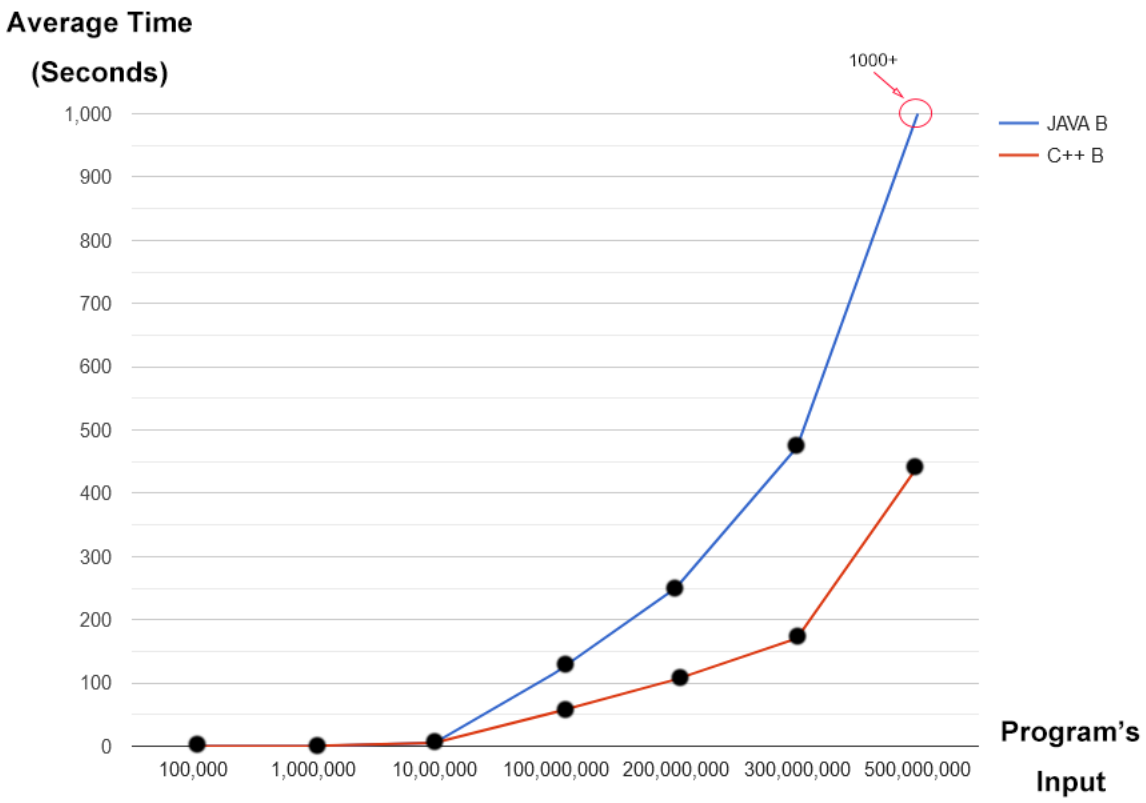


Program B

The B.java and B.cpp programs are doing a totally different thing than the A programs. They create a lot of alive objects and between them some garbage, almost 70 per cent of the heap size (Otherwise for a large input over 500.000.000 a GC overhead exception is thrown). The memory allocation can be regarded as in a row. So programs follow the following steps, create garbage, create a lot of alive objects, again create garbage, from the alive objects "kill" the 20 per cent, create new alive objects, add more garbage and finally "kill" all the alive objects.

Below are indicated the metrics for both programs and the performance of the B.java program with the use of different garbage collectors.

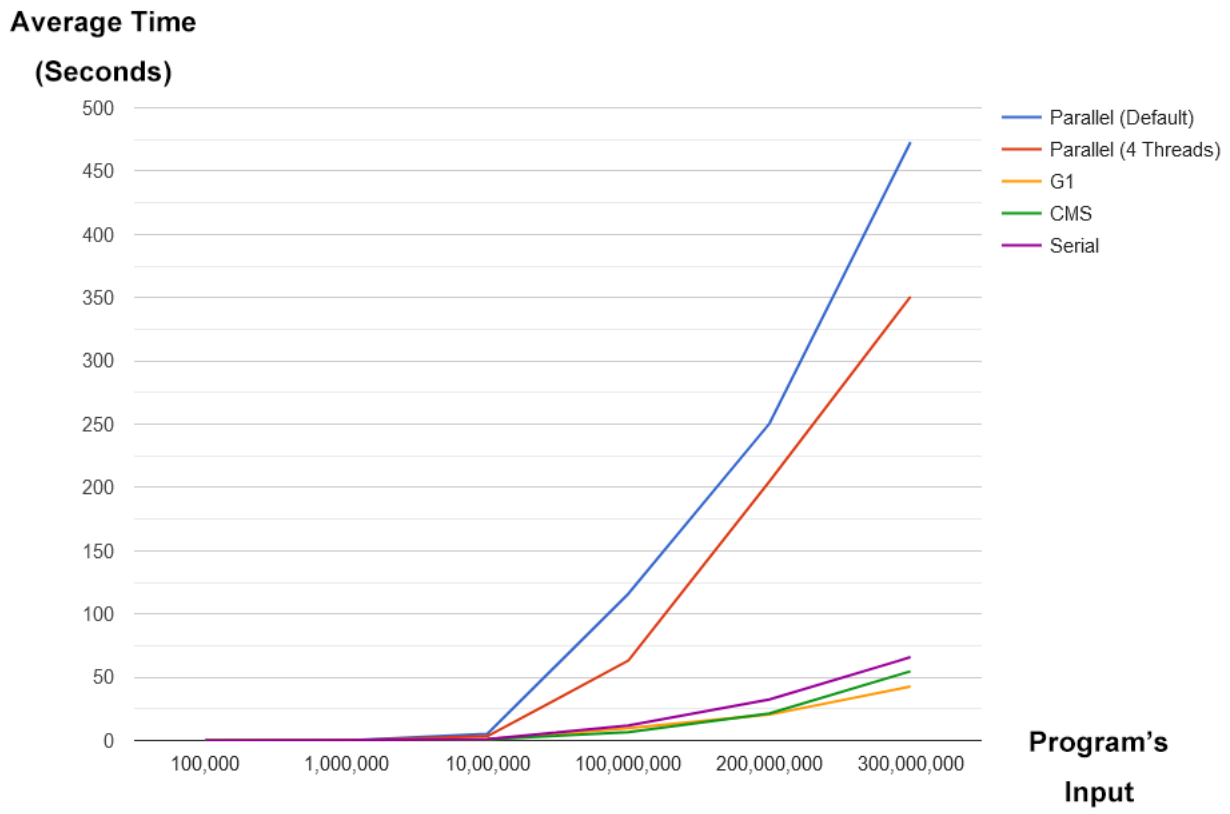
	Input	Java (B)	C++ (B)
1	100,000	0	0
2	1,000,000	0.52	0.4
3	10,00,000	5.22	5.45
4	100,000,000	116.08	53.75
5	200,000,000	250.51	105.81
6	300,000,000	472.83	170.64
7	500,000,000	1000	442.66



The metrics obviously show that as the program's input is getting increased the Java performance is getting decreased. On the other hand C++ terminates in a acceptable average time

Below are the metrics of the Java program with the use of different garbage collectors.

	INPUT	Parallel (Default)	Parallel (4 Threads)	CMS	G1	Serial
1	100,000	0	0	0	0	0
2	1,000,000	0.04	0.03	0.38	0	0.39
3	10,00,000	5.22	3.31	1	0.7	1.14
4	100,000,000	116.08	63.2	9.86	6.6	11.84
5	200,000,000	250.51	204.85	20.45	21.37	32.43
6	300,000,000	472.83	350.74	42.7	54.75	65.9



Garbage Collectors

Above on both A.java and B.java are used the following types of garbage collectors, "Parallel" which is the default Java8 GC, "Parallel" with the option **-XX:ParallelGCThreads=4** enabled, "G1", "CMS" and "Serial".

As we can notice as many threads we assign to the Parallel GC the performance of the program is increased. We can also notice that CMS, G1 and Serial have a stable performance with small difference for big inputs. But there is a very interesting point for the G1 garbage collector for the program A.java which for the input of 10.000.000 the performance of the program decreased dramatically. This is mostly about the way the G1 works, it tries to determine where are the alive objects in the heap, but at the A program there were a lot of garbage between the alive objects, so the sweeping phase is really slow.

The **NewSize** and **MaxNewSize** control the new generation's minimum and maximum size. The bigger the new generation, the less often the garbage collector will get triggered. The size of the new generation relative to the old generation is controlled by **NewRatio**. For instance, setting **-XX:NewRatio=3** means that the ratio between the old and new is 1:3. By default **NewRatio** for the JVM is 2: the old generation occupies 2/3 of the heap while the new generation occupies 1/3. The larger new generation can accommodate many more short-lived objects, decreasing the need for slow major collections. The old generation is still sufficiently large enough to hold many long-lived objects.

Versions, Hardware and Commands

Java: 1.8.0_261

C++ (g++): 9.3.0

Ram: 16Gb DDR4

CPU: Intel i7-9750H (2.6GHz)

java –> javac (A/B).java
java -XX:+Use**<TYPE>**GC -Xmx**<SIZE>**G -XX:NewRatio=**<RATIO>** (A/B) **<IN>**

cpp –> g++ -Wl,–stack,**<Stack Size>** (A/B).cpp -o (A/B)
./(A/B) **<IN>**

<TYPE>: Serial, Parallel, ParNew, G1
<SIZE>: 14
<Stack Size>: 5010612736 (In Bytes)
<RATIO>: 3
<IN>: 1000000
-XX:NewSize=size
-XX:MaxNewSize=size