Linnéuniversitetet
Kalmar Växjö

Report

Java Time Experiments

Author: Eric Enoksson
Semester: Spring 2018
Course: 1DV507

Introduction

The report examines the difference in performance between String and StringBuilder concatenations. It also examines the performance difference between insertion sort and merge sort, sorting integers and strings. The tests were done on an i5-8250U, 8 GB 2400MHz DDR4.

String concatenations

I created two methods: one for standard String concatenations, and one for StringBuilder concatenations with the appending string as a parameter. I ran each of them 5 times and computed an average.

For the short string concatenations, I just sent a string containing "a" to both methods. The length of the final string is the same as the amount of concatenations. For the long concatenations I used a sting of 80 a's, computing the average by dividing the concatenated string length with the appending string length.

String concatenation tests

String concatenation: short string. Duration: 1 sec

Run nr	String length / Concatenations
1	77360
2	77739
3	77450
4	75866
5	78087
average	**77300**

Table 1: String concatenations of short string

String concatenation: long string. Duration: 1 sec

Run nr	String length	Concatenations
1	489440	6118
2	490000	6125
3	505840	6323
4	479840	5998
5	499920	6249
average	**493008**	**6162**

Table 2: String concatenations of long string

StringBuilder concatenation tests

With the StringBuilder method, the test first concatenates for one second; then it measures the time it took to perform a toString() on the resulting string. After that, the test runs again, decreasing or increasing the time of the concatenation part to get closer to 1 sec in total.

I noticed re-occuring numbers, and numbers that were exactly twice or half of those numbers. This is most likely because the array insertion operations are really quick (and the time it takes for the insertions doesn’t increase as the array grows), while the resize becomes gradually slower as the array grows, resulting in the method pretty much always ending the run after a resize. That makes the time measuring increasingly inaccurate the longer the time measured and the faster the computer is. I introduced a certain time discrepancy allowance to help alleviate this problem.

StringBuilder concatenation: short string

Run nr	**String length / Concatenations**	**Time (ms)**
1	87089861	1043
2	80152319	1030
3	79599115	1027
4	75497471	1025
5	70227697	1022
average	**78513292**	**1029**

Table 3: StringBuilder concatenations of short string

StringBuilder concatenation: long string

Run nr	**String length**	**Concatenations**	**Time (ms)**
1	687865920	8598324	1042
2	687865920	8598324	1070
3	687865920	8598324	1079
4	687865920	8598324	967
5	687865920	8598324	1057
average	**687865920**	**8598324**	**1043**

Table 4: StringBuilder concatenations of long string

I noticed I had to lend JVM pretty much ALL of my memory to avoid out-of-memory errors. This because a fast computer will perform more actions leading to more memory usage.

Sorting algorithms

I changed the way I conducted my experience a couple of times. My initial thought was to modify the sorting algorithms slightly by adding a way of counting the nr of sorted objects. That became somewhat strange when sorting with merge sort since that algorithm splits the arrays first down to the size of 1 and then starts sorting the objects. Plus technically, it would add operations on the method I'm supposed to measure, even though they would probably not be too impactful.

I ended up doing it a cleaner way, with no change in the original code. This way takes a bit longer time, but is more accurate. My thinking was to construct new arrays until I found an array size that could be sorted in 1 second. I allowed a certain time discrepancy in order to find the size quicker and since the time the result wouldn't be completely accurate anyway since there are performance variations. If the array is sorted too quick, the array size increases; if it's sorted too slow, it array size is reduced. I used the same test method for both sorting methods since I could simply use an if-block to separate the tests. I wanted to find a way to let the array increase more if the target was far away and vice versa, so I calculated the measured time plus/minus the time taken (depending on if the sorting took longer or shorter than 1 second) and multiplied with a constant. I changed the constant depending on the sorting method used, since I knew the merge sort would in practice be significantly faster in all cases, meaning the array would be larger and so the changes would have to be larger as well.

Integer sorting

insertionSort integers

Run nr	**Array size**	**Time (ms)**
1	72720	979
2	72040	963
3	73560	1004
4	72520	972
5	73200	996
average	**72808**	**982**

Table 5: insertionSort integers

mergeSort integers

Run nr	**Array size**	**Time (ms)**

1	5605000	957
2	5609000	965
3	5585000	953
4	5617000	950
5	5593000	953
average	**5601800**	**955**

Table 6: mergeSort integers

insertionSort Strings

Run nr	**Array size**	**Time (ms)**
1	20820	1025
2	20780	985
3	19970	1006
4	19580	951
5	19440	950
average	**20118**	**983**

Table 7: insertionSort Strings

mergeSort Strings

Run nr	**Array size**	**Time (ms)**
1	1447000	1006
2	1355000	1032
3	1505000	1013
4	1478500	976
5	1325000	969
average	**1422100**	**999**

Table 8: mergeSort Strings

Reflection

It's really tricky to measure accurately with Java. Measuring algorithm performance doesn't seem to be Javas strong side, since JVM occasionally performs garbage collection, and code optimizations. A thing I noticed was that consecutive method executions calls was quicker than the first since JVM optimizes the code, but after doing a manual garbage collection, they weren't. That lead me to believe that the garbage collector throws away the optimizations as well, so it then raises a question if you should do them manually and wash away the optimizations or not run it manually and risk getting a garbage collection in the middle of a test.

StringBuilder vs String

The short explanation is that String concatenation performs more operations than StringBuilder concatenations.

Both a String and a StringBuilder consists of an array of characters. When a String object is created, it creates an array with the exact amount of slots needed, while the StringBuilder like an ArrayList creates an array with many empty spaces. Every time you concatenate strings, a new String object will be created, replacing the old one. Memory is allocated for the new object, data copied to the new object, and the object reference re-routed. Since it creates so many objects with low life span, it will also produce a lot of garbage wastefully using up memory, resulting in the need for the garbage collector to run more often.

A StringBuilder does not have to create a new object for every concatenation since it can fill the empty slots first. That results in significantly fewer memory allocations, copies, and less garbage when performing a large amount of concatenations, like in the experiment, which is why it is that much quicker.