Performance Experiments Report

B119172

November 24, 2017

1 Introduction

The purpose of this report is to demonstrate and analyze the results of performance experiments on the predator-prey model program.
The predator-prey model program is implemented in Java. By using Guava Stopwatch and Java VisualVM, this report presents investigations on program execution time and profiles the CPU and memory usage during runtime. The report then analyzes and identifies the main source of overhead. Further experiments on program optimizations were performed and presented.

2 Performance tests and analysis

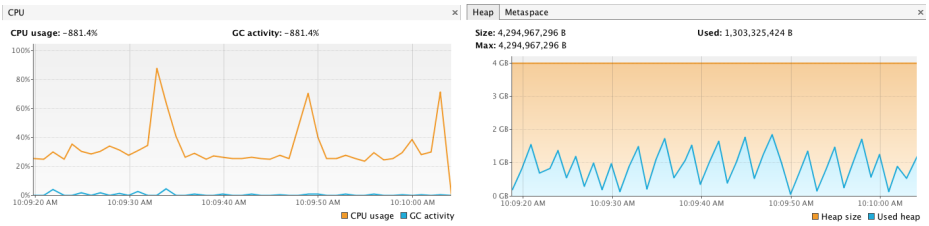
2.1 Performance

2.1.1 Context

All the experiments run on MacOS Sierra 10.12.6 with 1.6 GHz Intel Core i5 processor and 8 GB 1600 MHz DDR3 memory. Java version "1.8.0_73" with HotSpot 64-Bit Server VM was used. Java VisualVM was introduced as the profiling tool and Guava Stopwatch library was used to measure the execution time.

2.1.2 Profiling

Profiling by Java VisualVM for experiments on 500 x 500 landscape with 40 percent of land are presented below.In this experiment, output interval i was set to 400.



(a) CPU (b) Java Heap

Figure 1: Runtime Monitor Overview

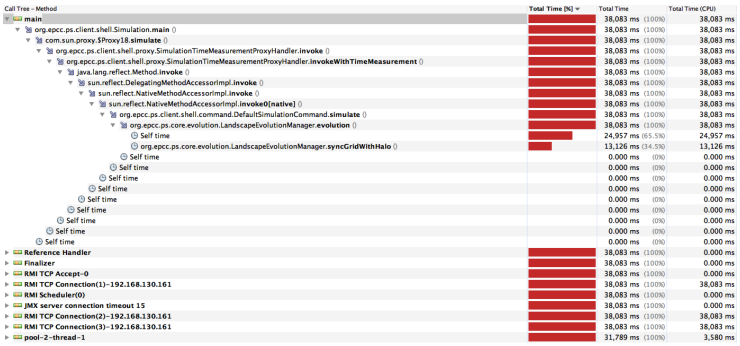


Figure 2: CPU Profiling

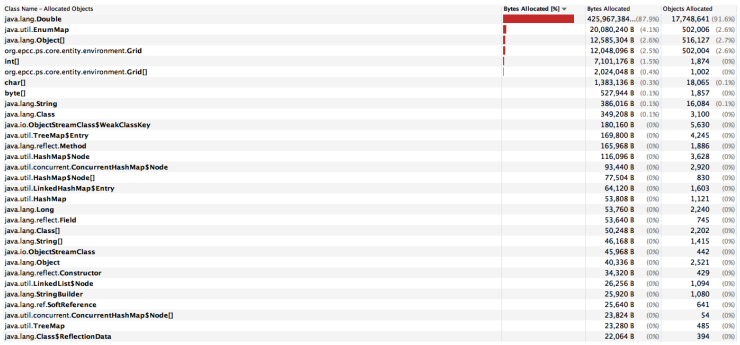


Figure 3: Memory Profiling

2.2 Analysis

2.2.1 Basic

In order to facilitate a better analysis, the execution time of the simulation is analyzed using the following simplified model.

$$T = T_{io} + \sum_{i=1}^{loop} \Delta t_i + T_{gc}$$

where:

- T_{io} is the cost of program input and output
- t_i is the execution time for simulation on iteration i
- T_{gc} is the cost of JVM Garbage Collection

According to the equation, there are three major factors affect the overall execution time. In this program , a thread pool is created for ppm output so T_{io} has less effect on overall run time. The simulation procedure modeled by t_i has a large impact on the total execution time and T_{gc} also makes a difference in T. This report mainly focused on the analysis of t_i and T_{gc}.

2.2.2 CPU

As is illustrated in Figure 1 (a), CPU usage fluctuates at around 25% and fall after a sudden rise to about 85% three times.
After initialization, the program starts to simulate the evolution of the predator-prey model over loops. CPU usage remains level during every iteration since the workload between iterations are similar. Ppm outputs of current iteration are exported on given intervals which leads to the sudden increase of CPU usage.
According to Figure 2, thread pool 2 is used to output the ppm file and the total time consumed in thread pool 2 is less than the simulation process. This indicates that the T_{io} does not have a great impact on the total execution time. Execution Time is mainly consumed by t_i and it represents the time used in the evolution function of the program.
The evolution function computes new densities of hare and puma on every grids of the landscape and update the density maintained by a Map structure of the Grid class. After that, a Grid array represents the landscape with a water halo for intermediate status record are synchronized with current grids on the landscape. The main overhead of the program for now lies in the update procedure of densities.

2.2.3 Memory

As is shown in Figure 1, there are many JVM GC happened during runtime. From Figure 3, large amounts of Double objects are used during execution. Creation of Double instances leads to the increase of GC time. The usage of so many Double objects is considered as the major overhead for now.
By analyzing the code of the program, all the Double instances are used in the Grid class which uses a Map with Double as its value to maintain the density of hare and puma. When updating the density after computations on each iteration, the put method of Map is used and new instances of Double are created to represent the density after evolution. Implementations of the update method leads to the numerous initialization of Double objects which not only affect the t_i but also results in more JVM GC and increase the T_{gc}.

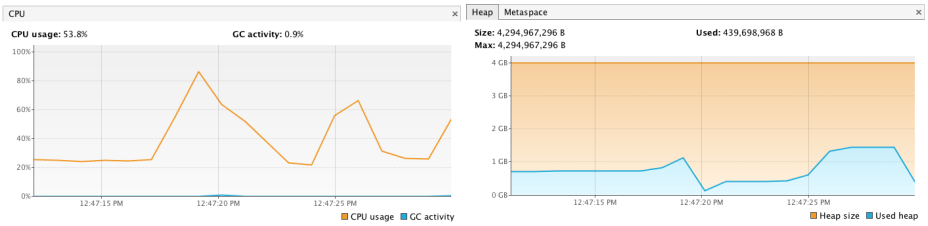
2.3 Optimization

2.3.1 Solution

Based on the analysis, a new way to manage densities on Grid class is introduced. A primitive double array is used to maintain the densities of hare and puma. When update on densities over iterations, only the value in the array is altered, which avoid the creation of Double objects.

2.3.2 Verification

After optimization as described above, experiments on 500 x 500 landscape with 40 percents of land are presented below.In this experiment, output interval i was set to 400.
As is demonstrated in Figure 4 and Figure 6, with less creation of Double instances, the memory usage has imporved and less GC was triggered. This contributes to the reduction of t_i with less object creation and Map operation and T_{gc} with less GC triggered. From Figure 5, the computation time is reduced. Further experiments on different problem size are repeated 10 times and the results are presented in Figure 7 to compare the difference between the previous code and optimized code.



(a) CPU (b) Java Heap

Figure 4: Runtime Monitor Overview

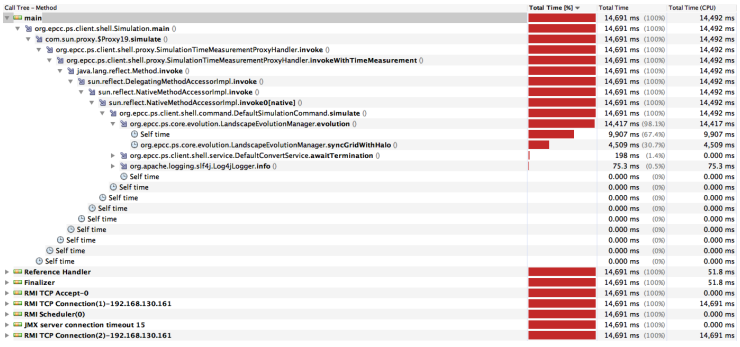


Figure 5: CPU Profiling

Class Name – Allocated Objects	Bytes Allocated [%]	Bytes Allocated	Objects Allocated
char[]		160,315,784... (31.9%)	5,344,567 (41%)
int[]		100,057,048... (19.9%)	358,576 (2.8%)
java.lang.String		56,282,112 B (11.2%)	2,345,088 (18%)
java.text.DecimalFormatSymbols		45,497,024 B (9.1%)	710,891 (5.5%)
double[]		22,089,024 B (4.4%)	503,513 (3.9%)
java.util.Formatter$FormatSpecifier		21,316,240 B (4.2%)	532,906 (4.1%)
java.util.Formatter$FixedString		17,061,432 B (3.4%)	710,893 (5.5%)
java.lang.Object[]		15,929,760 B (3.2%)	362,337 (2.8%)
java.lang.StringBuilder		12,796,656 B (2.5%)	533,194 (4.1%)
org.epcc.ps.core.entity.environment.Grid		12,048,096 B (2.4%)	502,004 (3.9%)
java.util.regex.Matcher		11,391,552 B (2.3%)	177,993 (1.4%)
java.util.Formatter$FormatString[]		8,535,008 B (1.7%)	177,992 (1.4%)
java.util.Formatter$Flags		8,526,656 B (1.7%)	532,916 (4.1%)
java.util.Formatter		5,695,776 B (1.1%)	177,993 (1.4%)
org.epcc.ps.core.entity.environment.Grid[]		2,024,048 B (0.4%)	1,002 (0%)
byte[]		386,064 B (0.1%)	969 (0%)
java.lang.Class		349,328 B (0.1%)	3,101 (0%)
java.lang.reflect.Method		166,320 B (0%)	1,890 (0%)
java.util.HashMap$Node		109,920 B (0%)	3,435 (0%)
java.util.concurrent.ConcurrentHashMap$Node		95,008 B (0%)	2,969 (0%)
java.util.HashMap$Node[]		68,944 B (0%)	675 (0%)
java.util.LinkedHashMap$Entry		64,360 B (0%)	1,609 (0%)
java.util.LinkedList$Node		59,280 B (0%)	2,470 (0%)
java.lang.Double		59,064 B (0%)	2,461 (0%)
java.util.TreeMap$Entry		54,800 B (0%)	1,370 (0%)
java.io.ObjectStreamClass$WeakClassKey		54,016 B (0%)	1,688 (0%)
java.lang.reflect.Field		53,640 B (0%)	745 (0%)
java.util.HashMap		51,600 B (0%)	1,075 (0%)
java.lang.Class[]		47,976 B (0%)	2,131 (0%)
java.lang.String[]		40,912 B (0%)	1,277 (0%)
java.lang.Object		39,808 B (0%)	2,488 (0%)

Figure 6: Memory Profiling

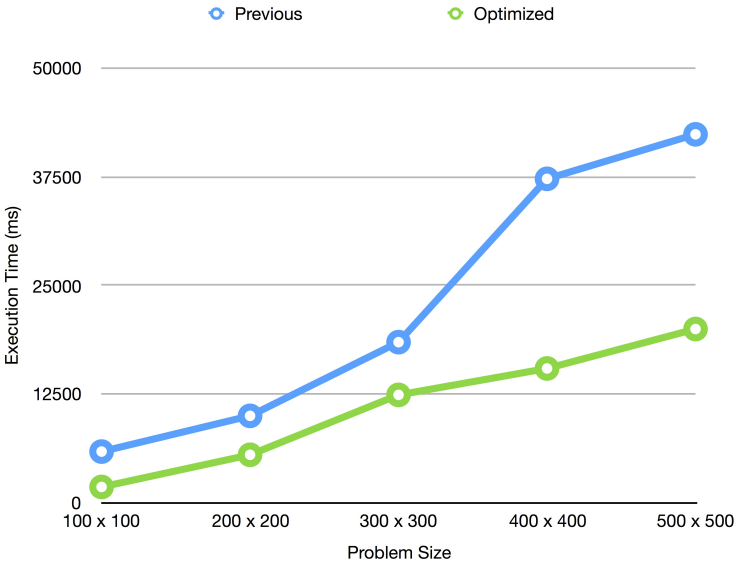


Figure 7: Execution time of previous and optimized code

3 Conclusion

This report presents experiments on performance investigations of the predator-prey model program. Different ways to manage program data makes a great difference on program performance. According to the analysis, numerous creation of objects in a Java program will result in an overhead on program execution and more JVM GC will be triggered which will also slow down the program. Under certain circumstance, with a properly designed data structure such as a primitive double array instead of a Map to maintain data that requires lots of updates during runtime will reduce the overhead on execution and JVM GC. It is important to choose a better data structure for program performance.