Breaking Java Badly - Programmatic Destructive Testing of Java Virtual Machines (Part One)

By
Mark Raley

Introduction

Java heap issues can be difficult to detect because of how the Java Virtual Machine (JVM) responds to memory pressure. It is designed to complete runnable tasks while being almost entirely indifferent to most performance considerations. It will translate and execute the most optimal version of the java programs it can, but the ultimate responsibility for efficiency resides in the architecture of the java software. Uninformed choices here can cause potentially serious performance and stability issues when insufficient resources are available to the JVM - troublesome issues that may not be readily identifiable from presented behaviors. The following series of tests, performed in batteries, explores this problem space. We will also present working python source code (in part three) that objectively quantifies some types of heap pressure in the tenured generation (compatible with OpenJDK 1.8).

A Simple Producer-Consumer Test

The memory stress test chosen is a simple producer-consumer setup. There is one producer thread paired with one consumer thread connected by a synchronized data container (for test one this is a Linked List, hereafter referred to as the queue). The producer adds numbers to the tail of the queue as rapidly as it can, while the consumer removes them from the head as fast as it can. Since only one of them can access the queue at a time, they must take turns. When the consumer sees the Nth number transit the queue it exits which completing the test and the execution time is recorded. If production and consumption remain balanced, the buffer remains small. If production exceeds consumption, however, the queue will fill.

There is one additional feature of this test, which limits the queue from growing past a specified maximum capacity limit (MCL). If the queue contains MCL elements already, the producer thread will wait on notification from the consumer. For this implementation, the linked list data structure is specifically chosen because of the number of interconnects between objects (each element has a forward and backward memory reference), which in turn increases the load on the active memory management subsystem of the JVM (the garbage collector).

Simple Producer-Consumer Memory Test One (Linked List)

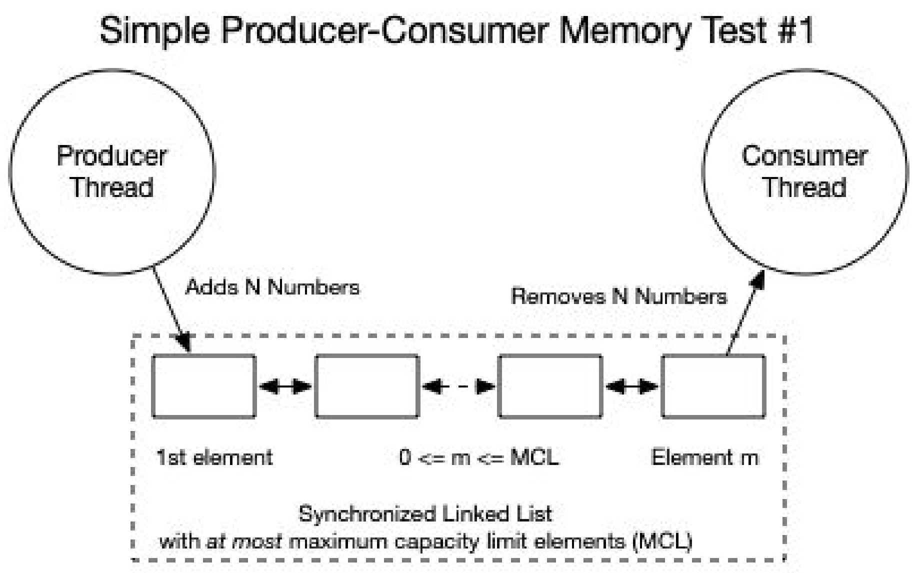

Figure One

The MCL setting value is then a configuration setting that limits the storage capacity of the linked list data structure. During test execution, the length of the queue (m) can vary anywhere from zero to the current MCL value (0 <= m <= MCL). The m value may remain small or grow as large as MCL, completely subject to how often each thread is scheduled, how long it executes without interruption, and how efficient each thread is relative to the other. As m increases (and the size of a Number remains more or less the same), it may be said that the memory pressure on the JVM is rising because it has no choice but to store more information in the heap. If the producer and the consumer threads operate in balance, m will remain stable. If not the MCL setting comes into play by causing the producer to wait until there is room for more numbers in the queue. But what if the MCL is set too high and the producer gets ahead of the consumer?

Testing Methodology

The maximum capacity limit (MCL) value is set to monotonically increasing values starting with one and four million and intervals of eight million thereafter. Because we are hunting potentially infrequent or rare events the test is run 72 times per MCL setting for a total of 7 * 72 = 504 separate test runs. 72 was chosen was because exploratory test sets of a dozen or so generated inconsistent results. It is worth noting the sufficiently rare events could require even more tests to isolate by this method. By proving some such rare events occur, and developing methods to detect these by other means, we can obviate the need for unnecessarily large test sets in the future. During each run, 768 million numbers are produced (inserted to one end) and consumed (removed from the other end) through the queue.

Results were collected from an 8 physical core 3.2Ghz Ryzen 1700 running OpenJDK 1.8.0_265 in a (VMWare virtualized) Ubuntu 20.04 LTS environment.

Full Analysis - Parallel Old - Test One

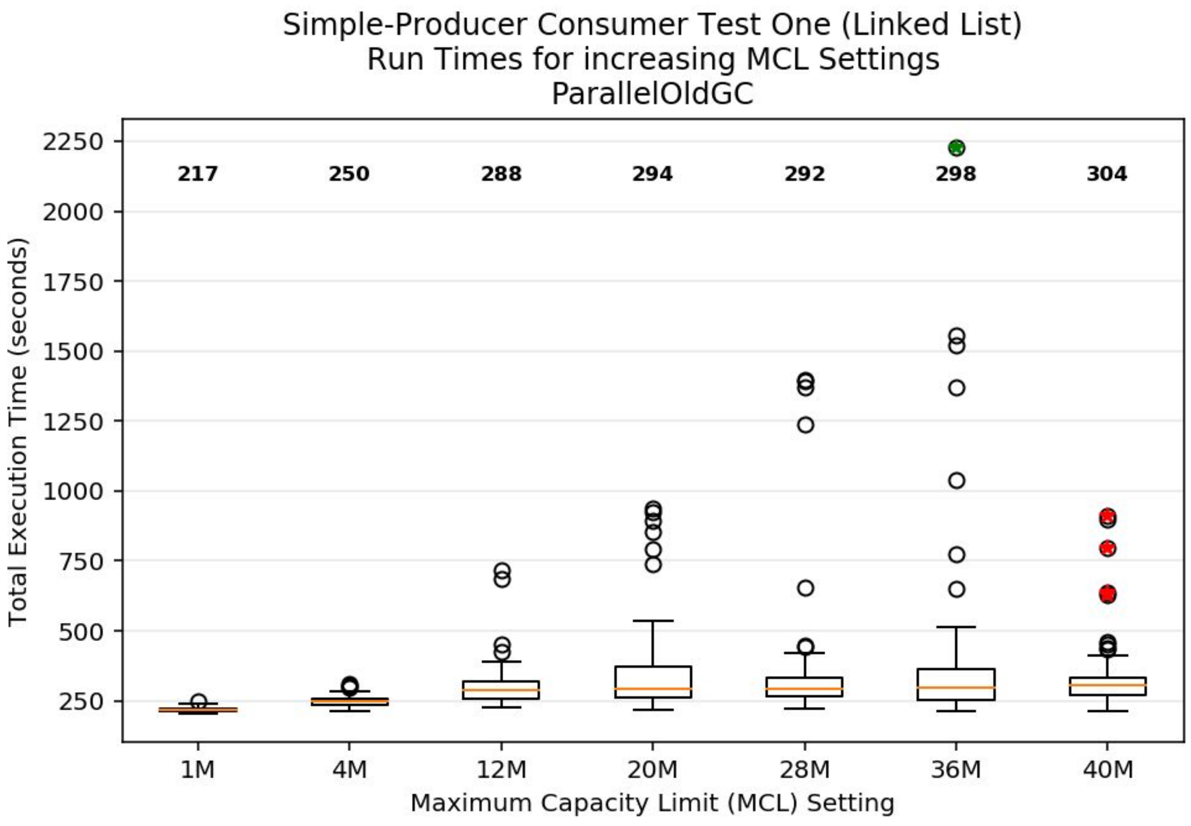

Figure Two

Qualitatively, we could characterize the shape of this data as follows at the 36M setting or lower;

1. Every group includes runs at or near best results (~205 seconds).
2. The median values between groups are not very different.
3. ~90% of the recorded run times are under 450 seconds.
4. Volatility increases as MCL settings increase.
5. About 7% from all groups are considered statistical outliers (marked as circles) and are all on the high side of each group

Limited testing over these data groups wouldn't necessarily be disconcerting at a glance. However at scale, and subject to the demands of a production environment, there could be potentially dozens of incidents of slowness three or four times beyond the expected results. Unaddressed, these might undermine user confidence in the platform and/or data-center. It also illustrates how collecting an insufficiency of samples can allow some types of issues to go undetected and perhaps contribute to unnecessary user dissatisfaction.

On the right of figure two there are several marked outliers;

1. There is one run where execution time exceeds 2000 seconds (green circle).
2. At and above the 40M MCL setting the JVM fails roughly 5% of the time (red circles).

The green marked outlier in the 36M column, taking as a series worse about 2250 seconds is worthy of special examination - we'll do this on such outliers in part three and where the real fun begins.

At the 40M data group the JVM sometimes terminates after appearing to stall. In my experience, these cases will alarm most users. We'll drill down into these as well in part three and beyond - but first, in part two, we'll use the Producer-Consumer test battery framework to take a short but intense look at the JDK's famed G1 collector.