Learning Resource

Simple & Effective G1 Garbage Collector Tuning Tips

By Ram Lakshmanan

G1 GC is an adaptive Garbage Collection algorithm, that has become the default GC algorithm since Java 9. We would like to share a few tips to tune G1 Garbage collector to obtain optimal performance.

Index

1. Maximum GC Pause time 04

2. Avoid setting young gen size 04

3. Remove Old arguments 04

4. Eliminating String Duplicates 04

5. Undertand Default Settings 05

6. Study GC Cases 06

   6.1. Full GC - Allocation Failure 07

   6.2. G1 Evacuation Pause/ Evacuation Failure 08

   6.3. G1 Humangous Allocation 09

   6.4. System.gc( ) 09

   6.5. Heap Dump initiated GC 10

1. Maximum GC Pause time

Consider passing '-XX:MaxGCPauseMillis' argument with your preferred pause time goal. This argument sets a target value for maximum pause time. G1 GC algorithm tries it's best to reach this goal.

2. Avoid Setting Young gen size

Avoid setting the young generation size to a particular size (by passing '-Xmn', '-XX:NewRatio' arguments). G1 GC algorithm modifies young generation size at runtime to meet its pause -time goals. If the young generation size is explicitly configured, then pause time goals will not be achieved.

3. Remove Old arguments

When you are moving from other GC algorithms (CMS, Parallel, ...) to G1 GC algorithm, remove all the JVM arguments related to the old GC algorithm. Typically passing old GC algorithms arguments to G1 will have no effect, or it can even respond in a negative way.

4. Eliminating String duplicates

Because of inefficient programming practices, modern applications waste a lot of memory. Here is a case study showing memory wasted by the Spring Boot framework. One of the primary reasons for memory wastage is the duplication of string. A recent study indicates that 13.5% of application's memory contains duplicate strings. G1 GC provides an option to eliminate duplicate strings when you pass the '-XX:+UseStringDeduplication' argument. You may consider passing this argument to your application if you are running on Java 8 update 20 and above. It has the potential to improve the overall application's performance. You can learn more about this property in this article.

5. Understand default settings

For tuning purposes, in the below table, we have summarized important G1 GC algorithm arguments and their default values:

G1 GC Argument	Description
-XX:MaxGCPauseMillis=200	Sets a maximum pause time value. The default value is 200 milliseconds.
-XX:G1HeapRegionSize=n	Sets the size of a G1 region. The value has to be power of two i.e. 256, 512, 1024,....It can range from 1MB to 32MB.
-XX:GCTimeRatio=12	Sets the total target time that should be spent on GC vs total time to be spent on processing customer transactions. The actual formula for determining the target GC time is [1 / (1 + GCTimeRatio)]. Default value 12 indicates target GC time to be [1 / (1 + 12)] i.e. 7.69%. It means JVM can spend 7.69% of its time in GC activity and remaining 92.3% should be spent in processing customer activity.
-XX:ParallelGCThreads=n	Sets the number of the Stop-the-world worker threads. If there are less than or equal to 8 logical processors then sets the value of n to the number of logical processors. Say if your server 5 logical processors then sets n to 5. If there are more than eight logical processors, set the value of n to approximately 5/8 of the logical processors. This works in most cases except for larger SPARC systems where the value of n can be approximately 5/16 of the logical processors.
-XX:ConcGCThreads=n	Sets the number of parallel marking threads. Sets n to approximately 1/4 of the number of parallel garbage collection threads (ParallelGCThreads).

-XX:G1OldCSetRegion ThresholdPercent=10	Sets a maximum pause time value. The Sets an upper limit on the number of old regions to be collected during a mixed garbage collection cycle. The default is 10 percent of the Java heap. value is 200 milliseconds.
-XX:G1ReservePercent=10	Sets the percentage of reserve memory to keep free. The default is 10 percent. G1 Garbage collectors will try to keep 10% of the heap to be free always.

6. Study GC Causes

One of the effective ways to optimize G1 GC performance is to study the causes triggering the GC and provide solutions to reduce them. Here are the steps to study the GC causes.

1. Enable GC log in your application. It can be enabled by passing following JVM arguments to your application during startup time:
   Up to Java 8:

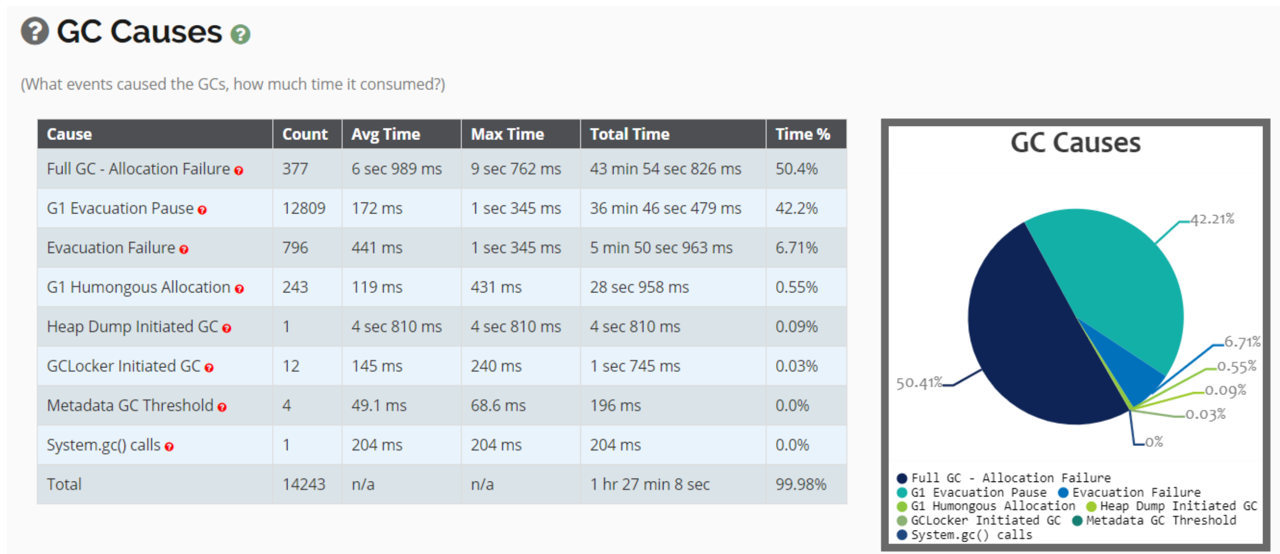
```
-XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:{file-path}
```

From Java 9 and above:

```
-XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:{file-path}
```

{file-path}: is the location where GC log file will be written

2. You can analyze the GC log file using free tools like GCeasy, Garbage Cat, HP Jmeter. These tools report the reasons that are triggering the GC activity. Below is the GC Causes table generated by GCeasy tool when G1 GC Log file was uploaded. Full analysis report can be found here.



GC Causes

(What events caused the GCs, how much time it consumed?)

Cause	Count	Avg Time	Max Time	Total Time	Time %
Full GC - Allocation Failure	377	6 sec 989 ms	9 sec 762 ms	43 min 54 sec 826 ms	50.4%
G1 Evacuation Pause	12809	172 ms	1 sec 345 ms	36 min 46 sec 479 ms	42.2%
Evacuation Failure	796	441 ms	1 sec 345 ms	5 min 50 sec 963 ms	6.71%
G1 Humongous Allocation	243	119 ms	431 ms	28 sec 958 ms	0.55%
Heap Dump Initiated GC	1	4 sec 810 ms	4 sec 810 ms	4 sec 810 ms	0.09%
GCLocker Initiated GC	12	145 ms	240 ms	1 sec 745 ms	0.03%
Metadata GC Threshold	4	49.1 ms	68.6 ms	196 ms	0.0%
System.gc() calls	1	204 ms	204 ms	204 ms	0.0%
Total	14243	n/a	n/a	1 hr 27 min 8 sec	99.98%

Fig: G1 GC causes (excerpt from GCeasy report)

Below are the solutions to address each of them.

6.1. Full GC – Allocation Failure

Full GC – Allocation failures happen for two reasons:

- Application is creating too many objects that can't be reclaimed quickly enough.
- When heap is fragmented, direct allocations in the Old generation may fail even when there is a lot of free space.

Here are the potential solutions to address this problem:

1. Increase the number of concurrent marking threads by setting '-XX:ConcGCThreads' value. Increasing the concurrent marking threads will make garbage collection run fast.

2. Force G1 to start marking phase earlier. This can be achieved by lowering '-XX:InitiatingHeapOccupancyPercent' value. Default value is 45. It means the G1 GC marking phase will begin only when heap usage reaches 45%. By lowering the value, the G1 GC marking phase will get triggered earlier so that Full GC can be avoided.

3. Even though there is enough space in a heap, a Full GC can also occur due to lack of a contiguous set of space. This can happen because of a lot of humongous objects present in the memory (refer to section '6.3. G1 Humongous Allocation' in this article). A potential solution to solve this problem is to increase the heap region size by using the option '-XX:G1HeapRegionSize' to decrease the amount of memory wasted by humongous objects.

6.2. G1 Evacuation Pause or Evacuation Failure

When you see G1 Evacuation pause, then G1 GC does not have enough memory for either survivor or promoted objects or both. The Java heap cannot expand since it is already at its max. Below are the potential solutions to fix this problem:

1. Increase the value of the '-XX:G1ReservePercent' argument. Default value is 10%. It means the G1 garbage collector will try to keep 10% of memory free always. When you try to increase this value, GC will be triggered earlier, preventing the Evacuation pauses.

2. Start the marking cycle earlier by reducing the '-XX:InitiatingHeapOccupancy Percent'. The default value is 45. Reducing the value will start the marking cycle earlier. GC marking cycles are triggered when heap's usage goes beyond 45%. On the other hand, if the marking cycle is starting early and not reclaiming, increase the '-XX:InitiatingHeapOccupancyPercent' threshold above the default value.

3. You can also increase the value of the '-XX:ConcGCThreads' argument to increase Application is creating too many objects that can't be reclaimed quickly enough. the number of parallel marking threads. Increasing the concurrent marking threads When heap is fragmented, direct allocations in the Old generation may fail even will make garbage collection run fast.

4. If the problem persists you may consider increasing JVM heap size (i.e. -Xmx).

6.3. G1 Humongous Allocation

Any object that is more than half a region size is considered a "Humongous object". If the regions contain humongous objects, space between the last humongous object in the region and the end of the region will be unused. If there are multiple such humongous objects, this unused space can cause the heap to become fragmented.
Heap fragmentation is detrimental to application performance. If you see several Humongous allocations, please ncrease your '-XX:G1HeapRegionSize'. The value will be a power of 2 and can range from 1MB to 32MB.

6.4. System.gc()

When 'System.gc()' or 'Runtime.getRuntime().gc()' API calls are invoked from your application, stop-the-world Full GC events will be triggered. You can fix this problem through following solutions:

a. Search & Replace

This might be a traditional method :-), but it works. Search in your application code base for 'System.gc()' and 'Runtime.getRuntime().gc()'. If you see a match, then remove it. This solution will work if 'System.gc()' is invoked from your application source code. If 'System.gc()' is going to be invoked from your 3rd party libraries, frameworks, or through external sources then this solution will not work. In such circumstance, you can consider using the option outlined in #b and #c.

b. -XX:+DisableExplicitGC

You can forcefully disable System.gc() calls by passing the JVM argument –'XX:+Disable ExplicitGC' when you launch the application. This option will silence all the 'System.gc()' calls that are invoked from your application stack.

c. -XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses

You can pass '-XX:+ExplicitGCInvokesConcurrent' JVM argument. When this argument is passed, GC collections run concurrently along with application threads to reduce the lengthy pause time.

d. RMI

If your application is using RMI, you can control the frequency in which 'System.gc()' calls are made. This frequency can be configured using the following JVM arguments when you launch the application:

```
-Dsun.rmi.dgc.server.gcInterval=n

-Dsun.rmi.dgc.client.gcInterval=n
```

The default value for these properties in
JDK 1.4.2 and 5.0 is 60000 milliseconds (i.e. 60 seconds)
JDK 6 and later release is 3600000 milliseconds (i.e. 60 minutes)
You might want to set these properties to a very high value so that it can minimize the impact. For more details about 'System.gc()' calls and it's GC impact, you can refer to this article.

6.5 Heap Dump Initiated GC

'Heap dump Initiated GC' indicates that heap dump was captured from the application using tools like jcmd, jmap, profilers,... Before capturing the heap dump, these tools typically trigger full GC, which will cause long pauses.. We shouldn't be capturing heap dumps unless there is absolute necessity.

We hope you find this article useful.
Best wishes to tune your application
to get optimal performance!