GC Intelligence Report

g1gc_2.log

Duration: 2 hrs 13 min 30 sec

System Time greater than User Time

WARNING

In **5 GC event's**, 'sys' time is greater than 'usr' time. It's not a healthy sign. Read our recommendations to **reduce sys time**

Timestamp	User Time (secs)	Sys Time (secs)	Real Time (secs)
2025-10-13T18:15:06.231	0.22	0.29	0.01
2025-10-13T18:15:22.400	0.8	1.41	0.05
2025-10-13T18:15:25.620	1.61	1.67	0.07
2025-10-13T18:15:31.651	1.62	2.04	0.08
2025-10-13T18:16:14.395	2.39	2.58	0.16

Recommendations

(**CAUTION:** Please do thorough testing before implementing below recommendations.)

- **44 sec 494 ms** of GC pause time is triggered by **'G1 Evacuation Pause'** event. This GC is triggered when copying live objects from one set of regions to another set of regions. When Young generation regions are only copied then Young GC is triggered. When both Young + Tenured regions are copied, Mixed GC is triggered..

  Solution:
  1. Evacuation failure might happen because of over tuning. So eliminate all the memory related properties and keep only min and max heap and a realistic pause time goal (i.e. Use only -Xms, -Xmx and a pause time goal -XX:MaxGCPauseMillis). Remove any additional heap sizing such as -Xmn, -XX:NewSize, -XX:MaxNewSize, -XX:SurvivorRatio, etc.

  2. If the problem still persists then increase JVM heap size (i.e. -Xmx).

  3. If you can't increase the heap size and if you notice that the marking cycle is not starting early enough to reclaim the old generation then reduce -XX:InitiatingHeapOccupancyPercent. The default value is 45%. Reducing the value will start the marking cycle earlier. On the other hand, if the marking cycle is starting early and not reclaiming, increase the -XX:InitiatingHeapOccupancyPercent threshold above the default value.

  4. You can also increase the value of the '-XX:ConcGCThreads' argument to increase the number of parallel marking threads. Increasing the concurrent marking threads will make garbage collection run fast.

  5. 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. Note: G1 GC caps this value at 50%.

- **2 sec 884 ms** of GC pause time is triggered by **'G1 Humongous Allocation'** event. Humongous allocations are allocations that are larger than 50% of the region size in G1. Frequent humongous allocations can cause couple of performance issues:
  1. 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.
  2. Until Java 1.8u40 reclamation of humongous regions were only done during full GC events. Where as in the newer JVMs, clearing humongous objects are done in cleanup phase.

  Solution:
  You can increase the G1 region size so that allocations would not exceed 50% limit. By default region size is calculated during startup based on the heap size. It can be overriden by specifying '-XX:G1HeapRegionSize' property. Region size must be between 1 and 32 megabytes and has to be a power of two. Note: Increasing region size is sensitive change as it will reduce the number of regions. So before increasing new region size, do thorough testing.

- **181 ms** of GC pause time is triggered by **'Metadata GC Threshold'** event. This type of GC event is triggered under two circumstances:
  1. Configured metaspace size is too small than the actual requirement
  2. There is a classloader leak (very unlikely, but possible).

Solution:
You may consider setting '-XX:MaxMetaspaceSize' to a higher value. If this property is not present already please configure it. Setting these arguments to a higher value will reduce 'Metadata GC Threshold' frequency. If you still continue to see 'Metadata GC Threshold' event reported, then you need to inspect metaspace contents. Learn how to inspect metaspace contents from **this article.**

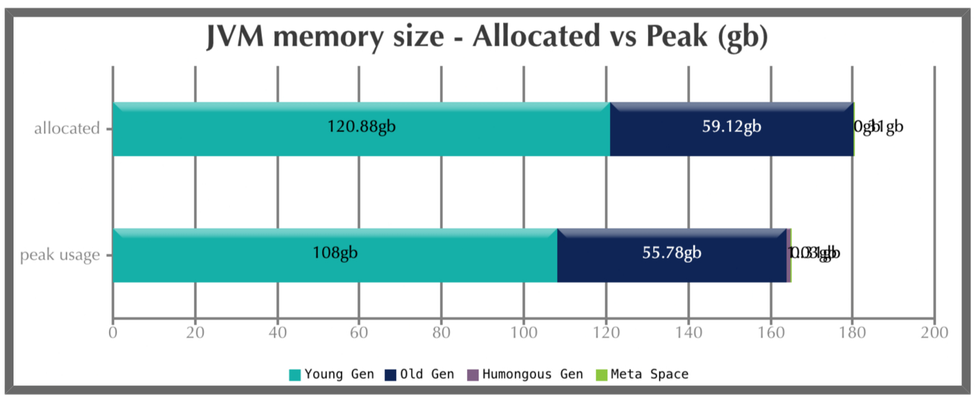
- It looks like you are using G1 GC algorithm. If you are running on Java 8 update 20 and above, you may consider passing **-XX:+UseStringDeduplication** to your application. It will remove duplicate strings in your application and has potential to improve overall application's performance. You can learn more about this property in **this article.**

- This application is using the G1 GC algorithm. If you are looking to tune G1 GC performance even further, here are the **important G1 GC algorithm related JVM arguments**

JVM memory size

(To learn about JVM Memory, click here)

Generation	Allocated	Peak
Young Generation	120.88 gb	108 gb
Old Generation	59.12 gb	55.78 gb
Humongous	n/a	1.03 gb
Meta Space	322.06 mb	315.93 mb
Young + Old + Meta space	180.31 gb	149.02 gb



Key Performance Indicators

(Important section of the report. To learn more about KPIs, click here)

1. **Throughput : 99.284%**

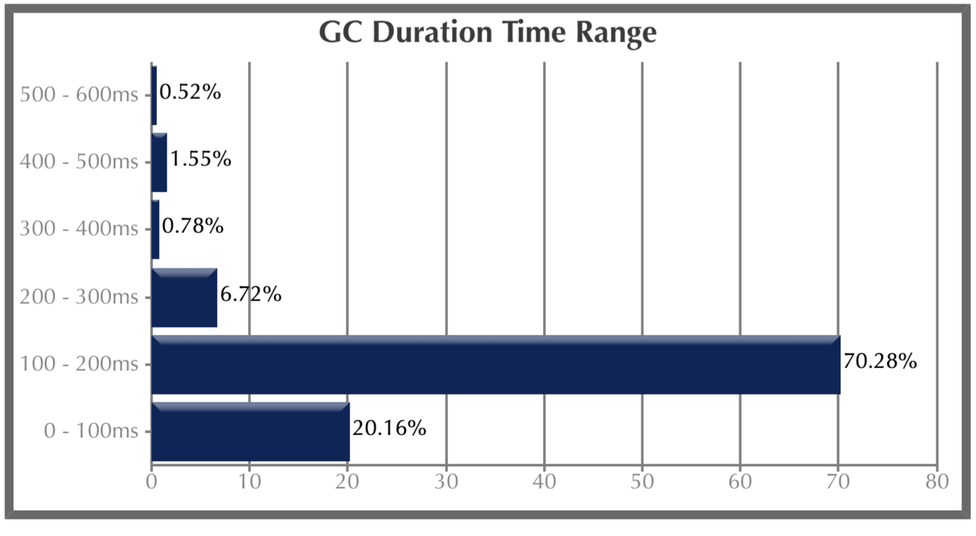
2. **CPU Time : 1 hr 13 min 18 sec**

3. **Latency:**

Avg Pause GC Time	**148 ms**
Max Pause GC Time	**571 ms**

GC Pause Duration Time Range :

Duration (ms) 100 ms Change	No. of GCs	Percentage
0 - 100	78	20.16%
100 - 200	272	70.28%
200 - 300	26	6.72%
300 - 400	3	0.78%



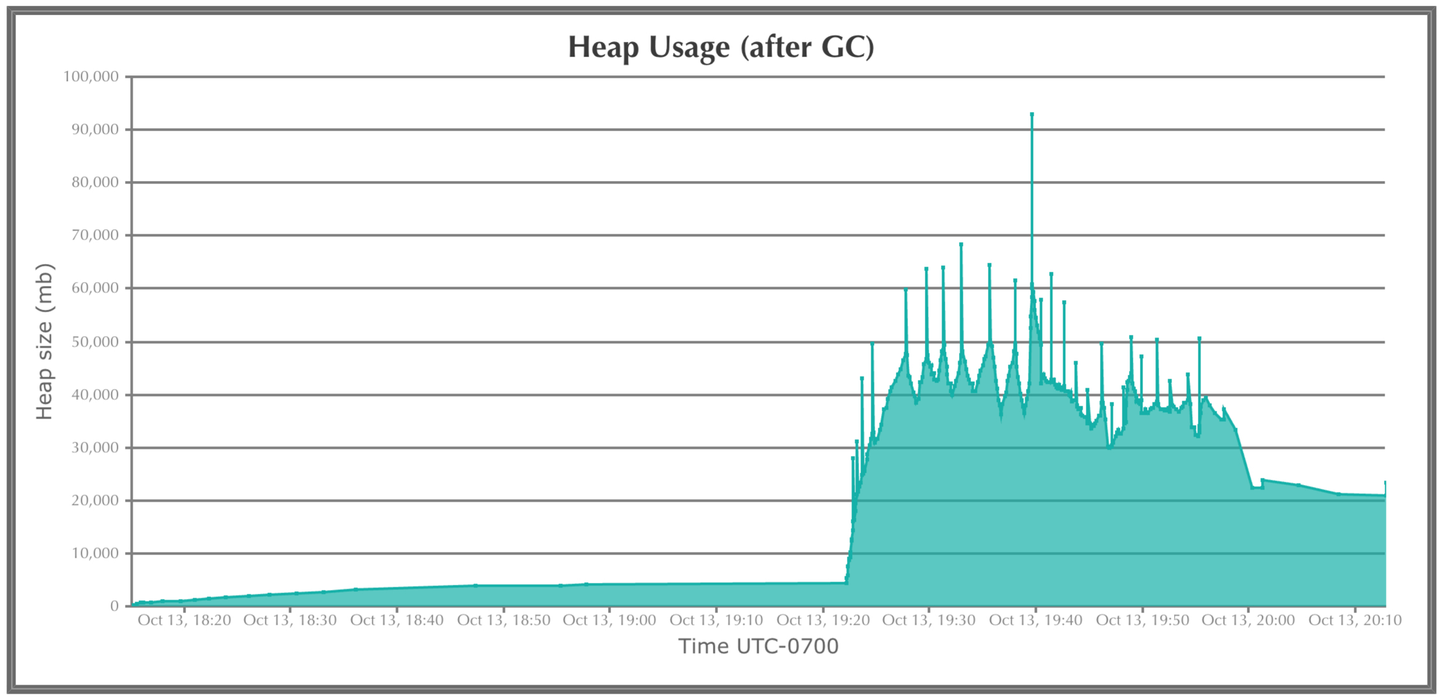
400 - 500	6	1.55%
500 - 600	2	0.52%

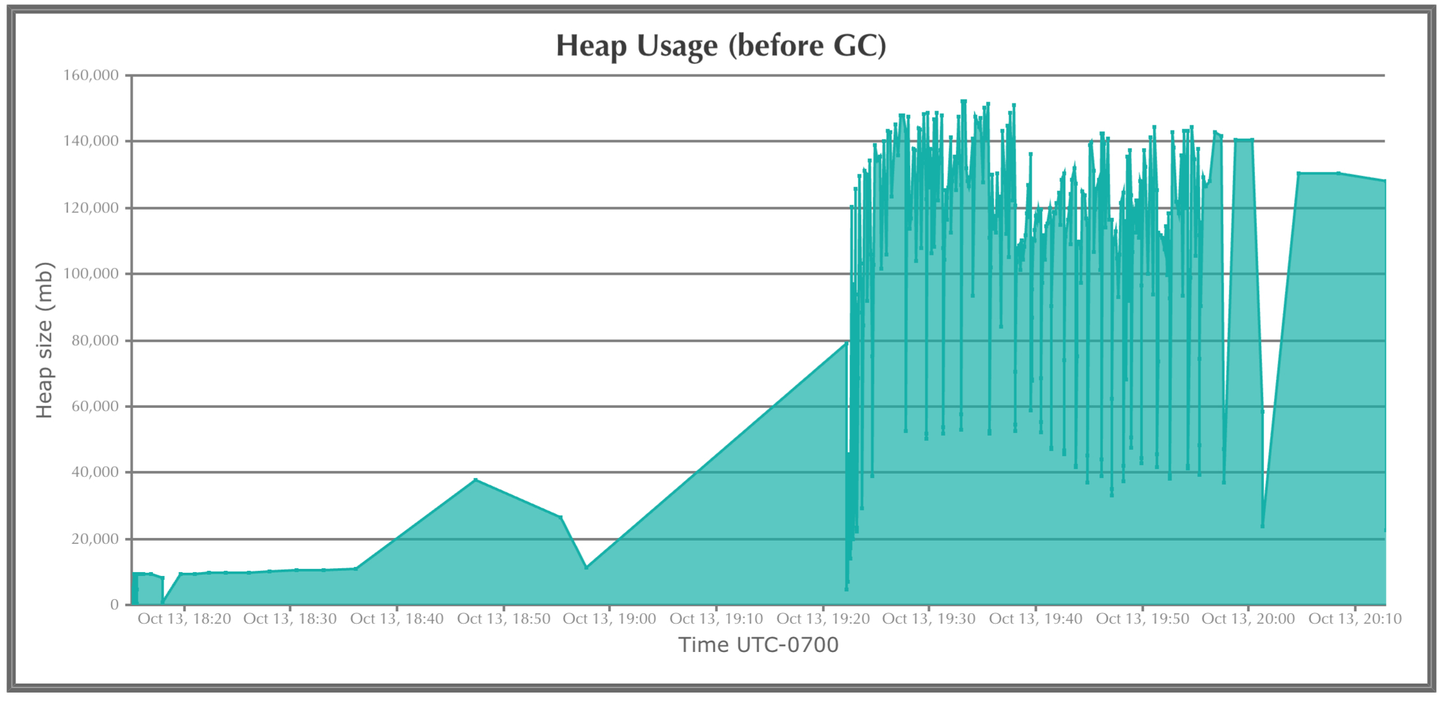
Analyze Specific Time Periods (Beta)

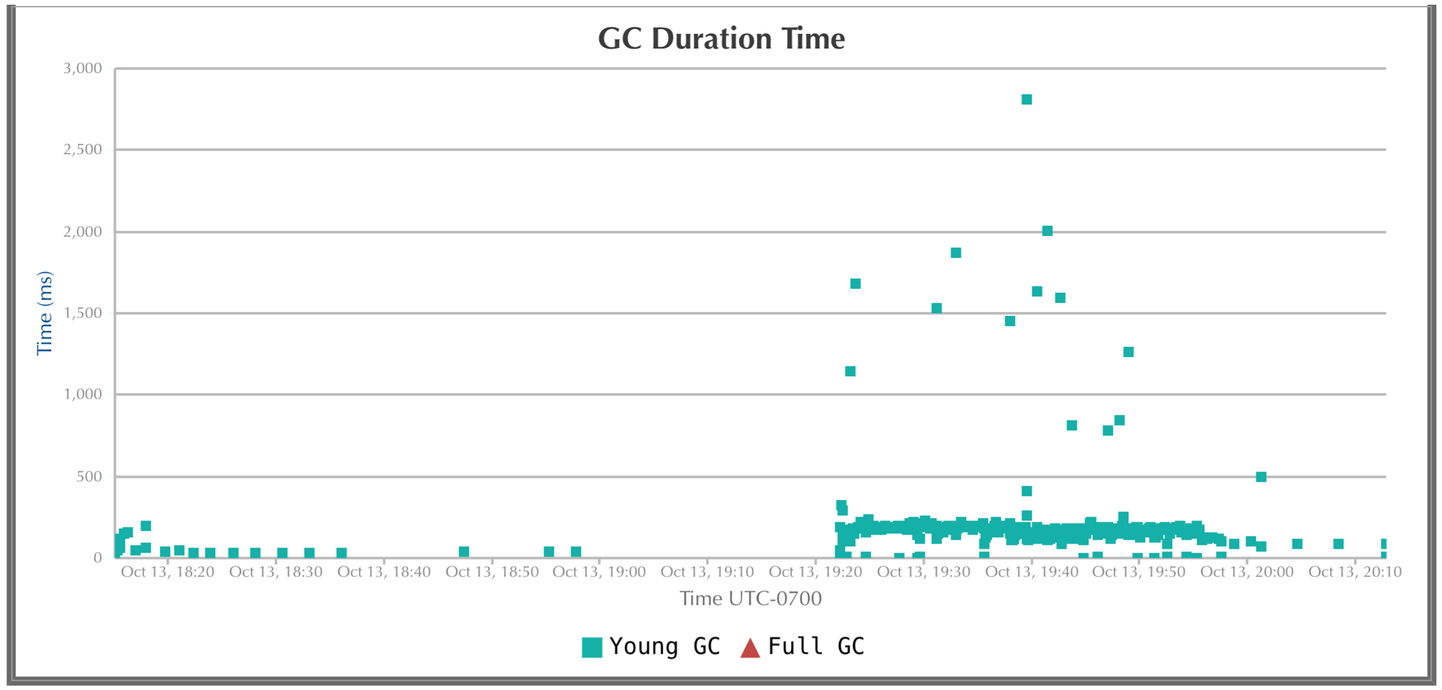
Select time range

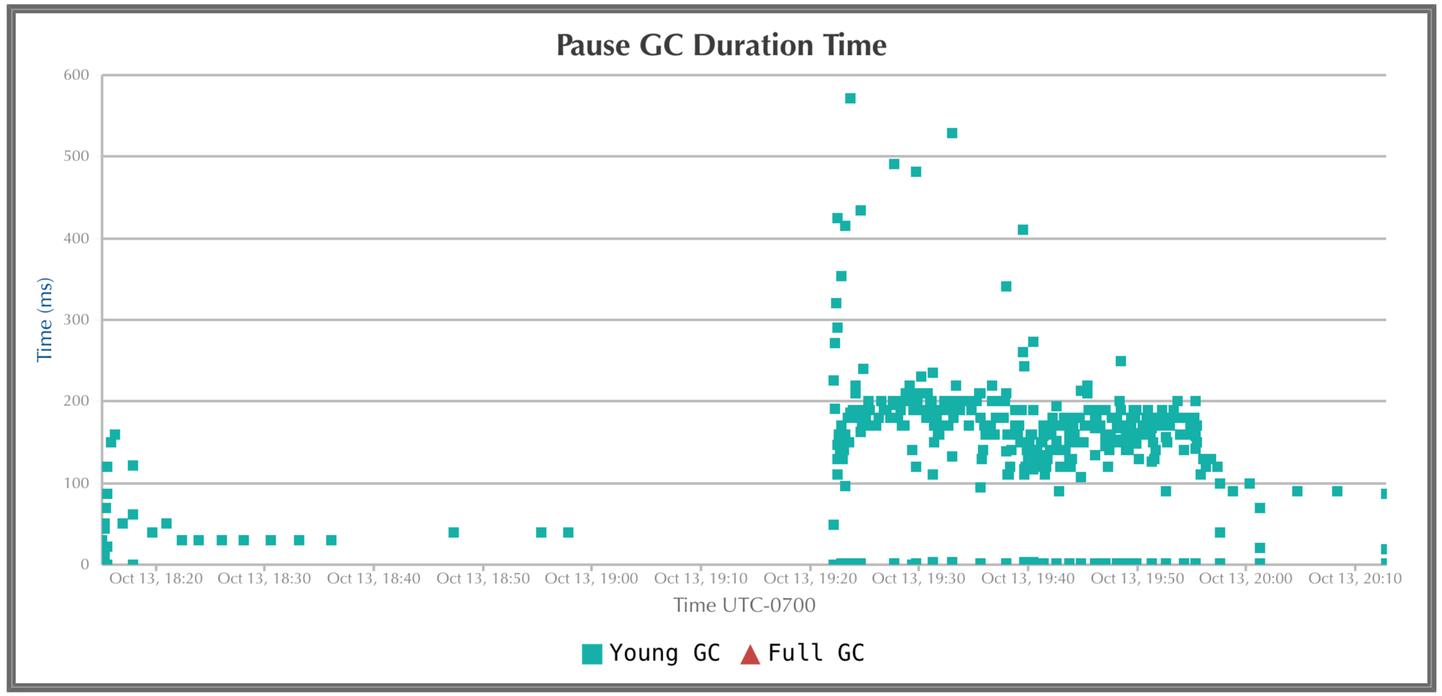
Reset

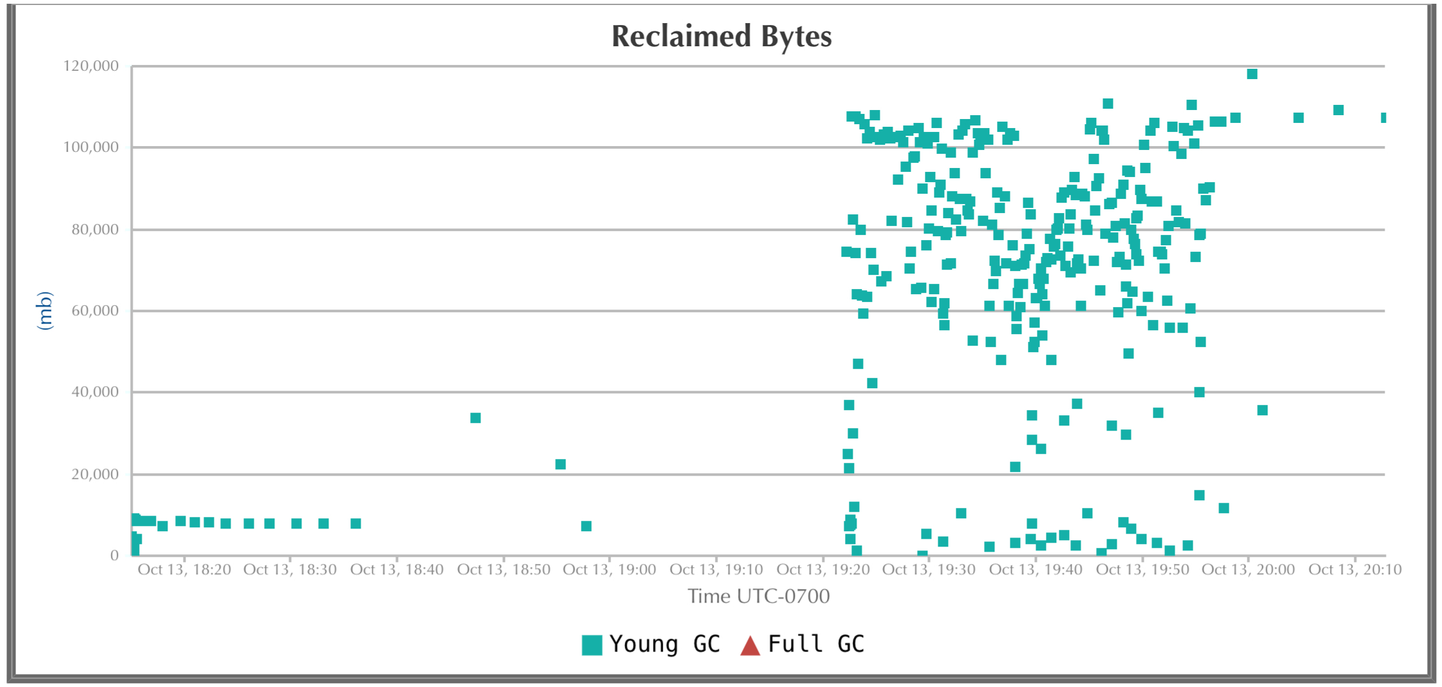
Interactive Graphs (How to zoom graphs?)

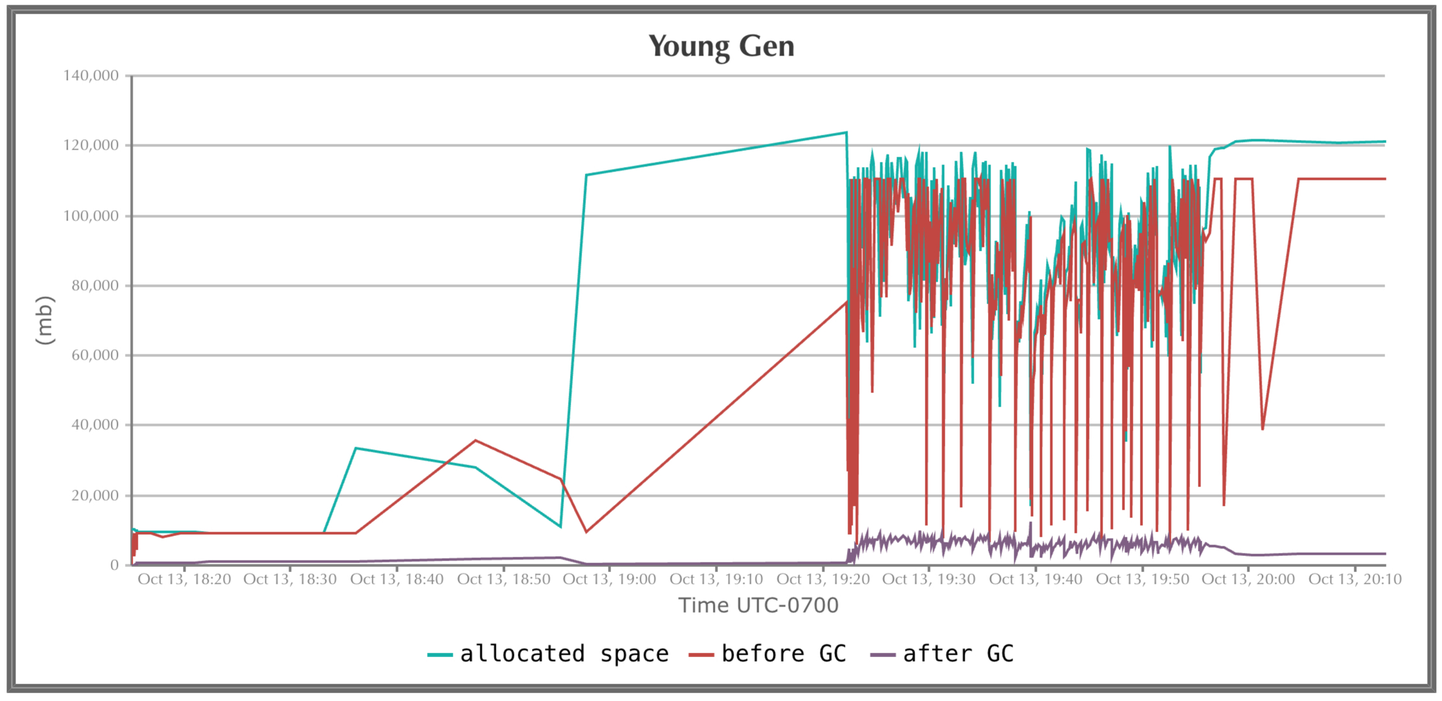


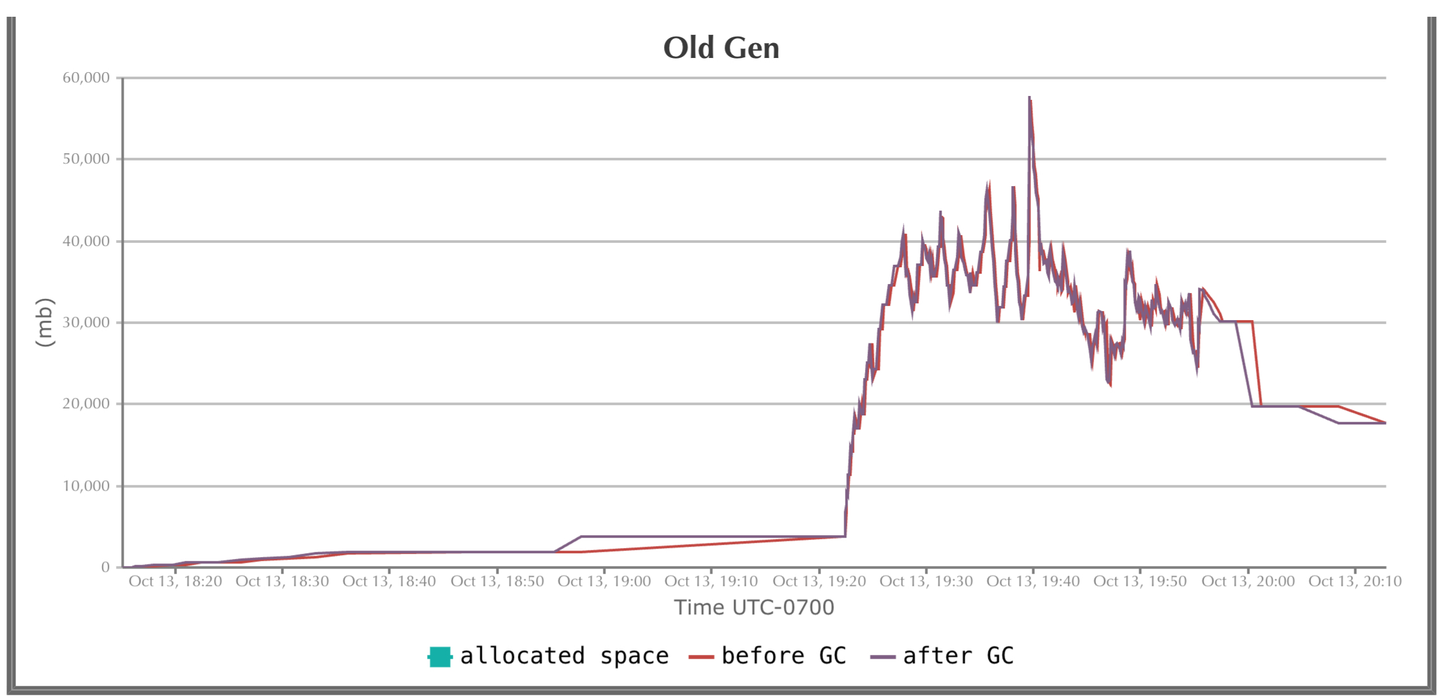


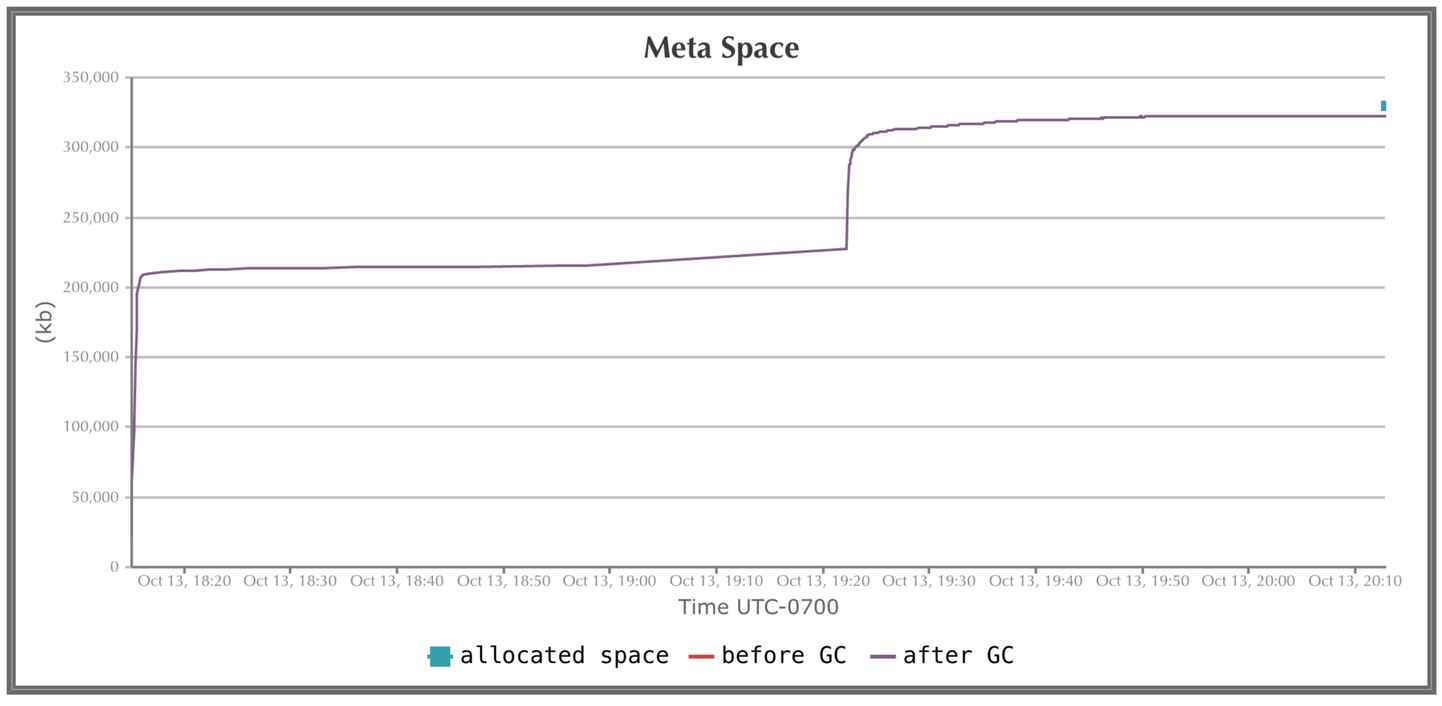

GC Duration Time
3,000
2,500
2,000
1,500
1,000
500
0
Time (ms)
Oct 13, 18:20
Oct 13, 18:30
Oct 13, 18:40
Oct 13, 18:50
Oct 13, 19:00
Oct 13, 19:10
Oct 13, 19:20
Oct 13, 19:30
Oct 13, 19:40
Oct 13, 19:50
Oct 13, 20:00
Oct 13, 20:10
Time UTC-0700
Young GC
Full GC

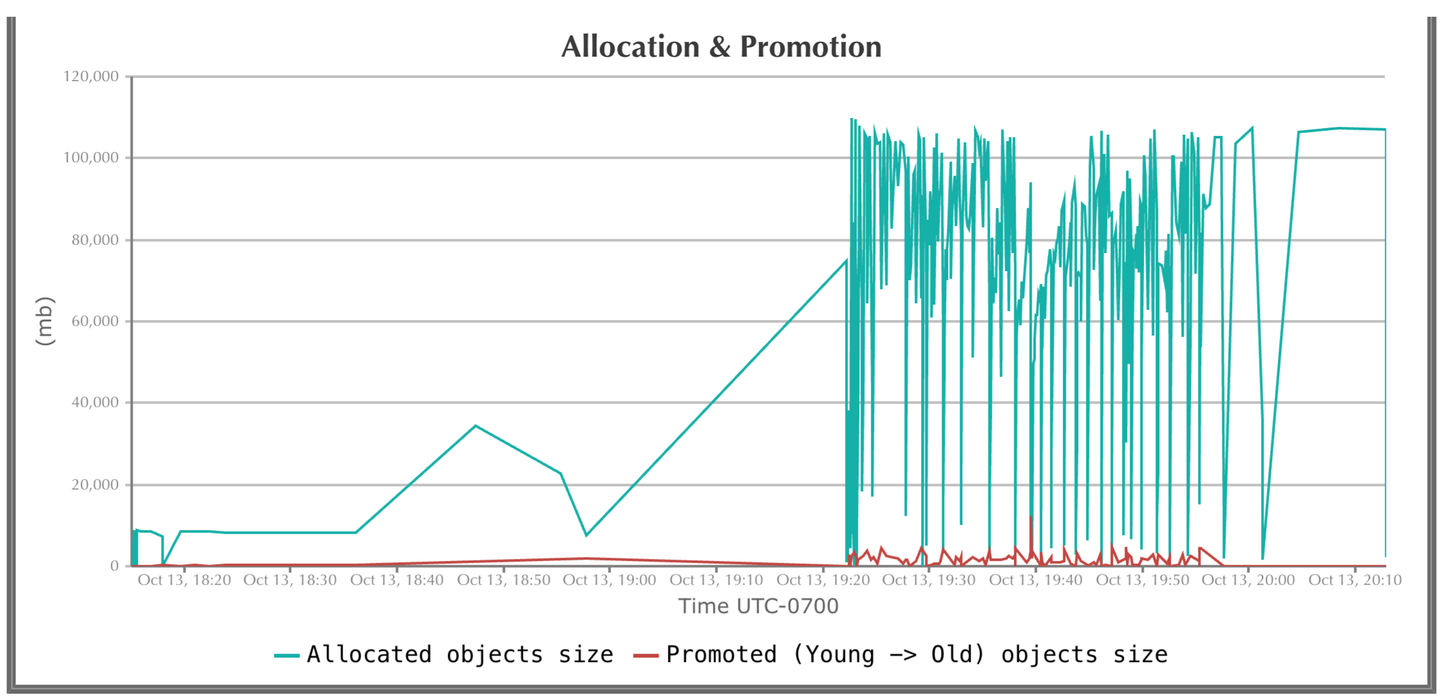

Pause GC Duration Time
600
500
400
300
200
100
0
Time (ms)
Oct 13, 18:20
Oct 13, 18:30
Oct 13, 18:40
Oct 13, 18:50
Oct 13, 19:00
Oct 13, 19:10
Oct 13, 19:20
Oct 13, 19:30
Oct 13, 19:40
Oct 13, 19:50
Oct 13, 20:00
Oct 13, 20:10
Time UTC-0700
Young GC
Full GC


Reclaimed Bytes
120,000
100,000
80,000
60,000
40,000
20,000
0
(mb)
Oct 13, 18:20
Oct 13, 18:30
Oct 13, 18:40
Oct 13, 18:50
Oct 13, 19:00
Oct 13, 19:10
Oct 13, 19:20
Oct 13, 19:30
Oct 13, 19:40
Oct 13, 19:50
Oct 13, 20:00
Oct 13, 20:10
Time UTC-0700
Young GC
Full GC

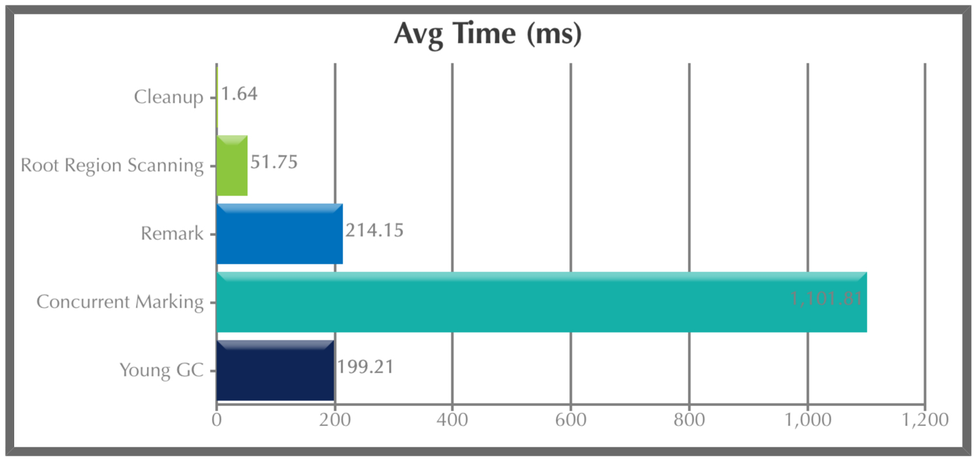

Young Gen
140,000
120,000
100,000
80,000
60,000
40,000
20,000
0
(mb)
Oct 13, 18:20
Oct 13, 18:30
Oct 13, 18:40
Oct 13, 18:50
Oct 13, 19:00
Oct 13, 19:10
Oct 13, 19:20
Oct 13, 19:30
Oct 13, 19:40
Oct 13, 19:50
Oct 13, 20:00
Oct 13, 20:10
Time UTC-0700
allocated space
before GC
after GC

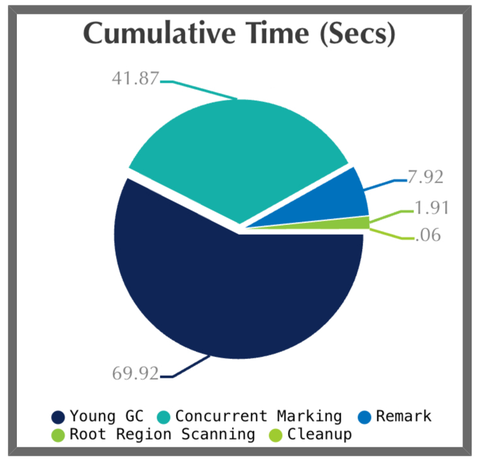

Old Gen
60,000
50,000
40,000
30,000
20,000
10,000
0
(mb)
Oct 13, 18:20
Oct 13, 18:30
Oct 13, 18:40
Oct 13, 18:50
Oct 13, 19:00
Oct 13, 19:10
Oct 13, 19:20
Oct 13, 19:30
Oct 13, 19:40
Oct 13, 19:50
Oct 13, 20:00
Oct 13, 20:10
Time UTC-0700
allocated space
before GC
after GC


Meta Space
350,000
300,000
250,000
200,000
150,000
100,000
50,000
0
(kb)
Oct 13, 18:20
Oct 13, 18:30
Oct 13, 18:40
Oct 13, 18:50
Oct 13, 19:00
Oct 13, 19:10
Oct 13, 19:20
Oct 13, 19:30
Oct 13, 19:40
Oct 13, 19:50
Oct 13, 20:00
Oct 13, 20:10
Time UTC-0700
allocated space
before GC
after GC



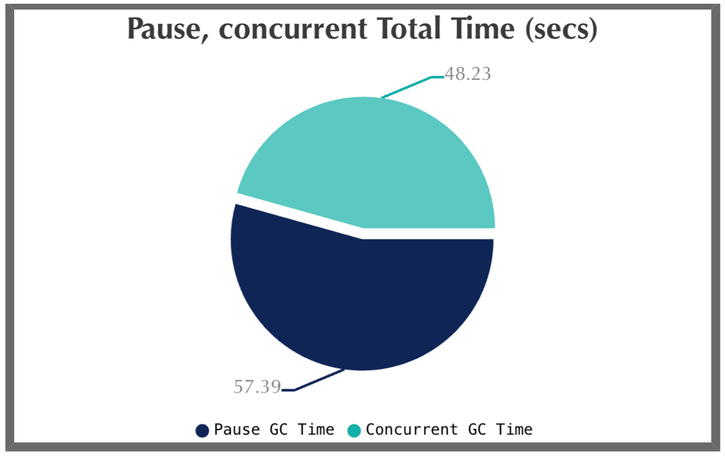
G1 Collection Phases Statistics

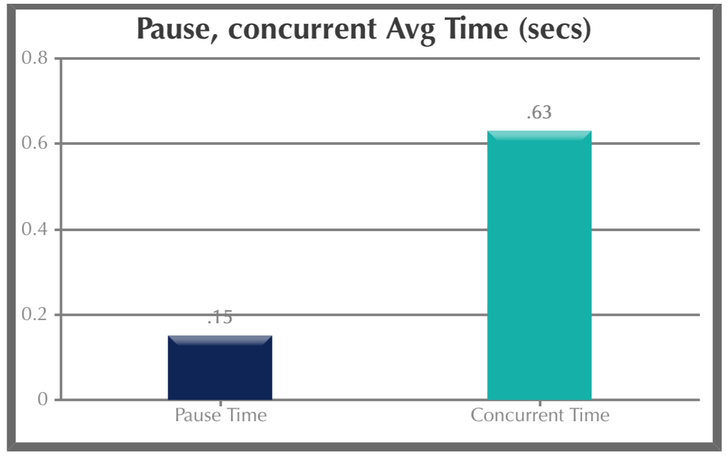




	Young GC	Concurrent Marking	Remark	Root Region Scanning	Cleanup
Total Time	1 min 9 sec 924 ms	41 sec 869 ms	7 sec 923 ms	1 sec 915 ms	60.8 ms
Avg Time	199 ms	1 sec 102 ms	214 ms	51.7 ms	1.64 ms
Std Dev Time	281 ms	738 ms	157 ms	31.2 ms	0.851 ms
Min Time	0	3.42 ms	1.84 ms	2.46 ms	0.00700 ms
Max Time	2 sec 810 ms	3 sec 240 ms	571 ms	125 ms	2.73 ms
Interval Time	20 sec 194 ms	3 min 11 sec 32 ms	3 min 16 sec 339 ms	3 min 16 sec 339 ms	3 min 16 sec 339 ms
Count	351	38	37	37	37

G1 GC Time





Pause Time

Total GC Pause Time	57 sec 385 ms
GC Pause Events	387.0
Avg GC Pause Time	148 ms
Std Dev GC Pause Time	82.2 ms
Min GC Pause Time	0.00700 ms
Max GC Pause Time	571 ms

Concurrent Time

Total Concurrent GC Time	48 sec 234 ms
Concurrent GC Events	76.0
Avg Concurrent GC Time	635 ms
Std Dev Concurrent GC Time	747 ms
Min Concurrent GC Time	3.71 ms
Max Concurrent GC Time	3 sec 309 ms

Object Stats

Total created bytes	20.01 tb
Total promoted bytes	188.63 gb
Avg creation rate	2.56 gb/sec
Avg promotion rate	24.11 mb/sec

CPU Stats

(To learn more about CPU stats, click here)

CPU Time:	1 hr 13 min 18 sec
User Time:	1 hr 12 min 9 sec
Sys Time:	1 min 8 sec 970 ms

Memory Leak

No major memory leaks.
(Note: there are 8 flavours of OutOfMemoryErrors. With GC Logs you can diagnose only 5 flavours of them(Java heap space, GC overhead limit exceeded, Requested array size exceeds VM limit, Permgen space, Metaspace). So in other words, your application could be still suffering from memory leaks, but need other tools to diagnose them, not just GC Logs.)

Consecutive Full GC

None.

Long Pause

None.

Safe Point Duration

(To learn more about SafePoint duration, click here)

	Total Time	Avg Time	% of total duration
Total time for which app threads were stopped	60.874 secs	0.016 secs	0.76 %
Time taken to stop app threads	6.992 secs	0.002 secs	0.087 %

Threads Affected By Allocation Stalls

(To learn more about Allocation Stall, click here)

Not Reported in the log.

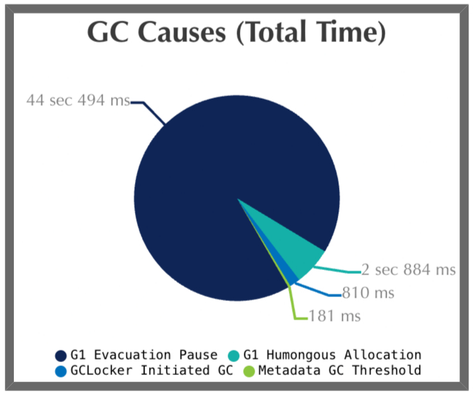
String Deduplication Metrics

Not Reported in the log.

GC Causes

(What events caused GCs & how much time they consumed?)

Cause	Count	Avg Time	Max Time	Total Time
G1 Evacuation Pause	272	164 ms	410 ms	44 sec 494 ms
G1 Humongous Allocation	23	125 ms	153 ms	2 sec 884 ms
GCLocker Initiated GC	5	162 ms	200 ms	810 ms
Metadata GC Threshold	5	36.2 ms	86.5 ms	181 ms



Tenuring Summary

Not reported in the log.

JVM Arguments

(To learn about JVM Arguments, click here)

Not reported in the log.

Export Data

Normalized GC Data