

GC Intelligence Report

📄 ee282dce-5c87-4859-9466-27bc96645343.txt

⌚ Duration: 5 sec 328 ms

💡 Recommendations

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

- ✓ 100 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:

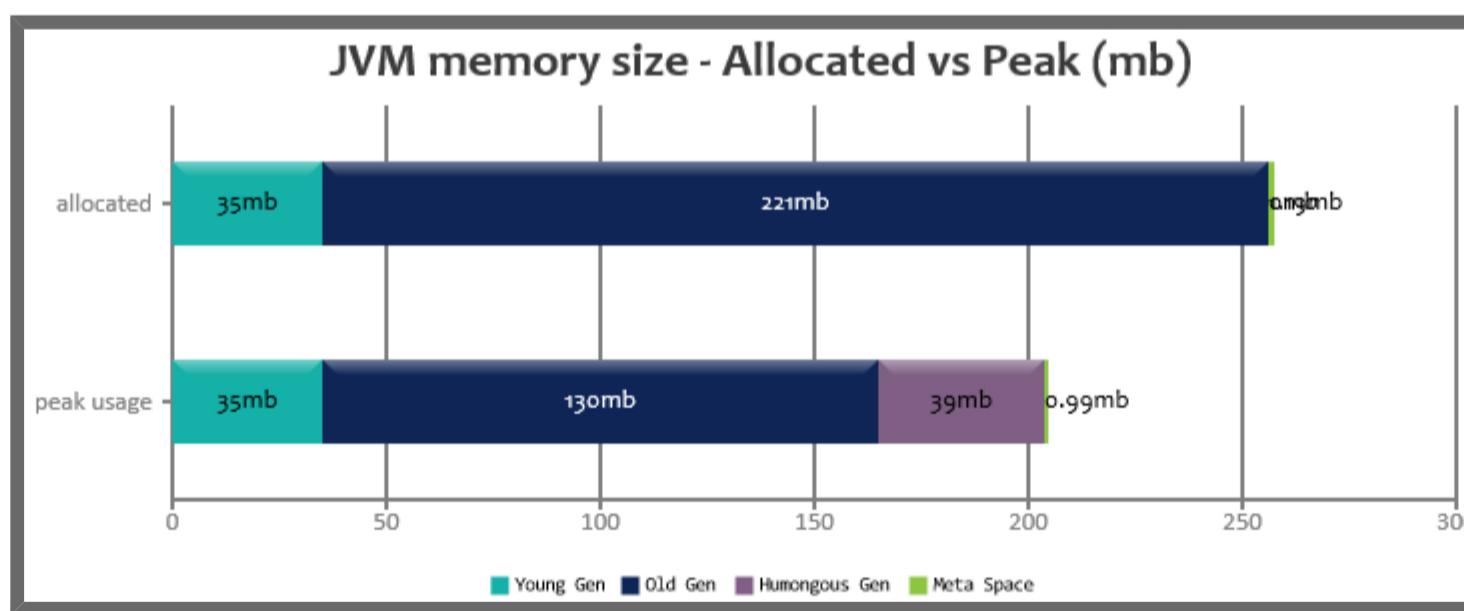
1. Evacuation failure might happen because of over tuning. So eliminate all the memory related properties and keep only min and max heap and a realistic pause time goal (i.e. Use only -Xms, -Xmx and a pause time goal -XX:MaxGCPauseMillis). Remove any additional heap sizing such as -Xmn, -XX:NewSize,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

- ✓ 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	35 mb	35 mb
Old Generation	221 mb	130 mb
Humongous	n/a	39 mb
Meta Space	1.19 mb	1,016 kb
Young + Old + Meta space	257.19 mb	200.99 mb



🔍 Key Performance Indicators

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

① Throughput : 98.116%

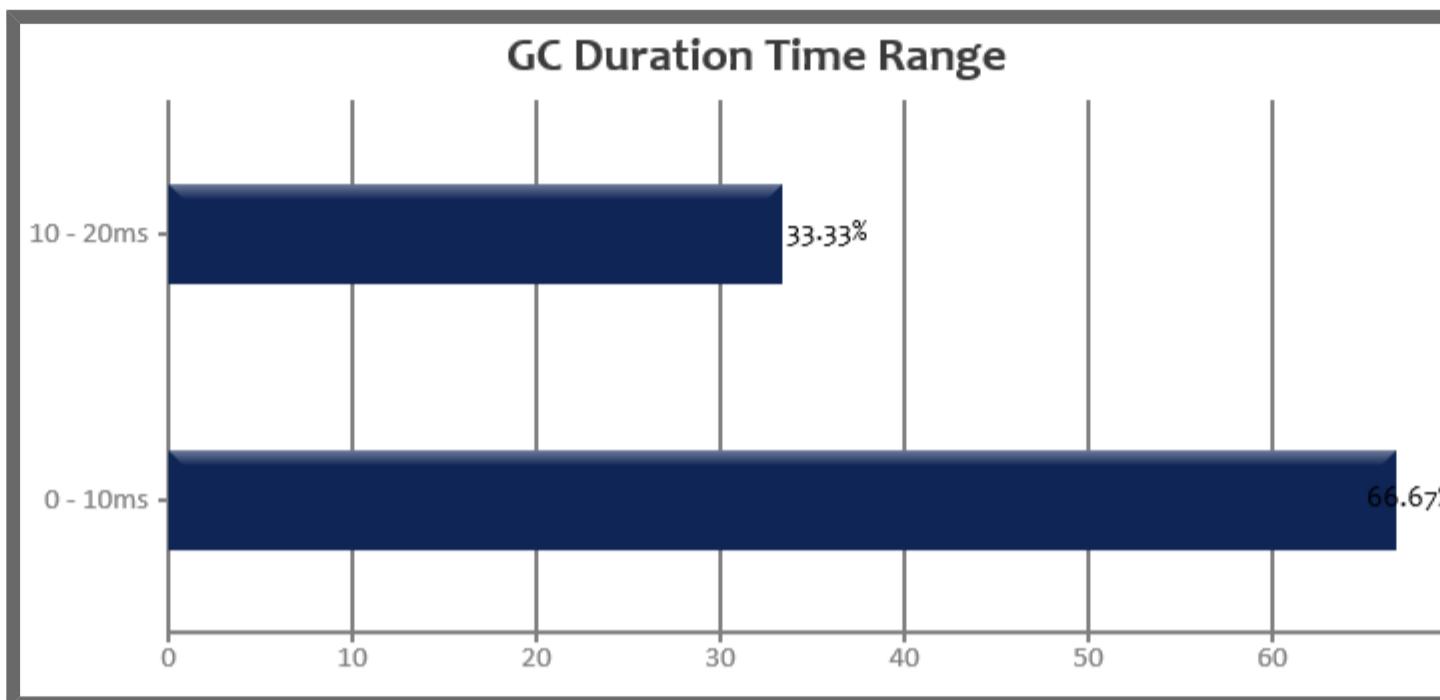
② CPU Time : 660 ms

③ Latency:

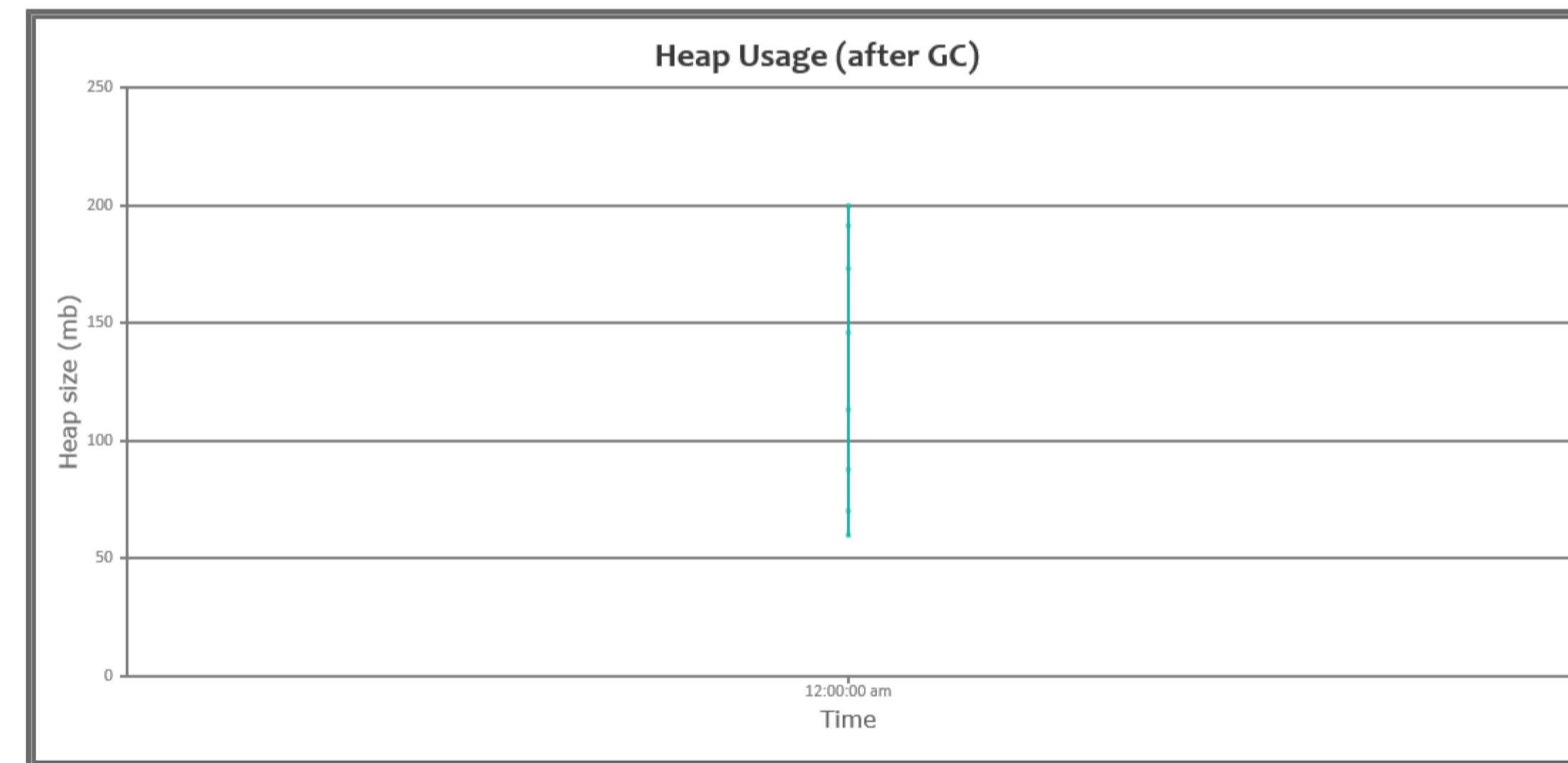
Avg Pause GC Time	11.2 ms
Max Pause GC Time	20.0 ms

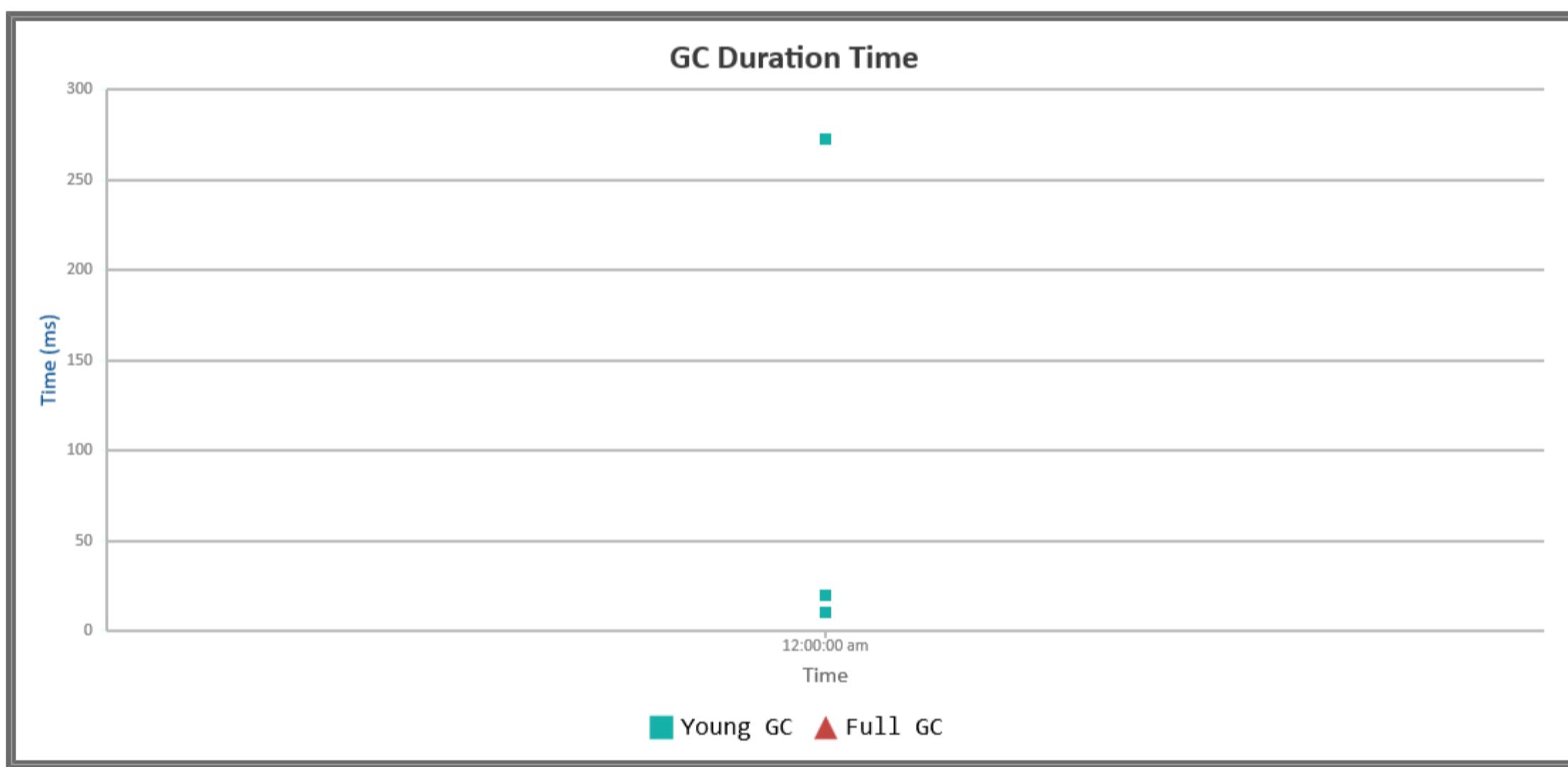
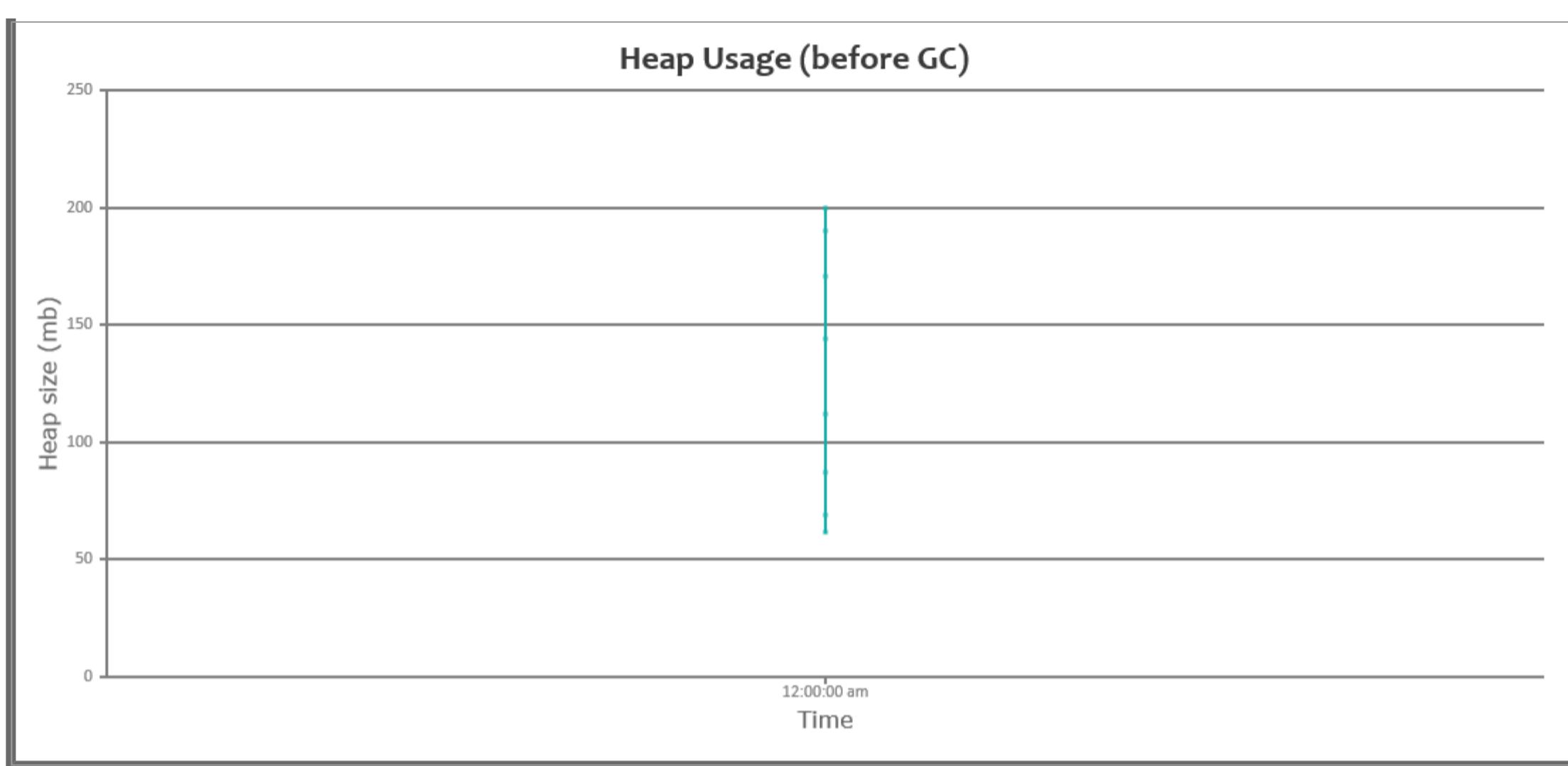
GC Pause Duration Time Range :

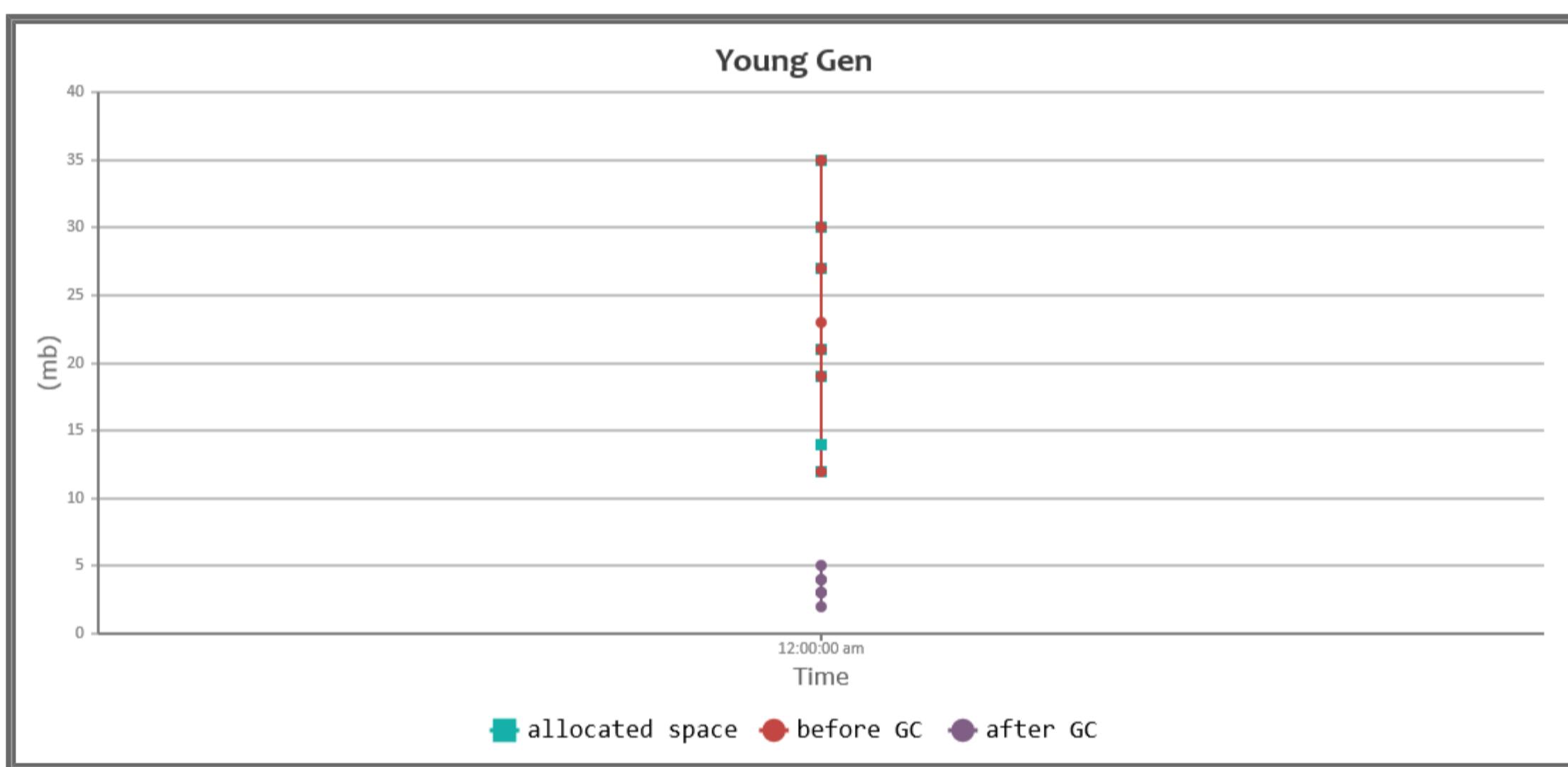
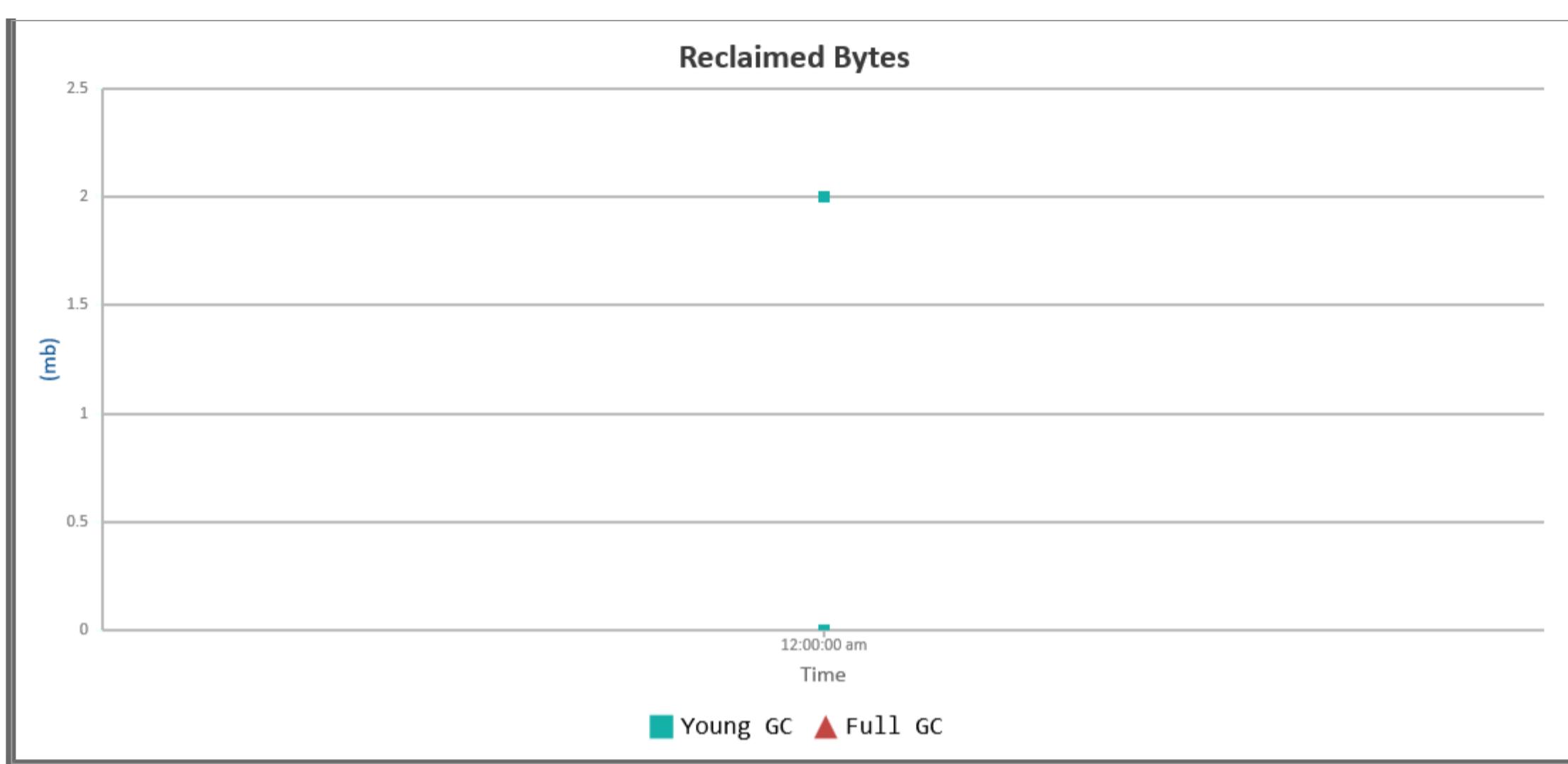
Duration (ms)	No. of GCs	Percentage
0 - 10	6	66.67%
10 - 20	3	33.33%

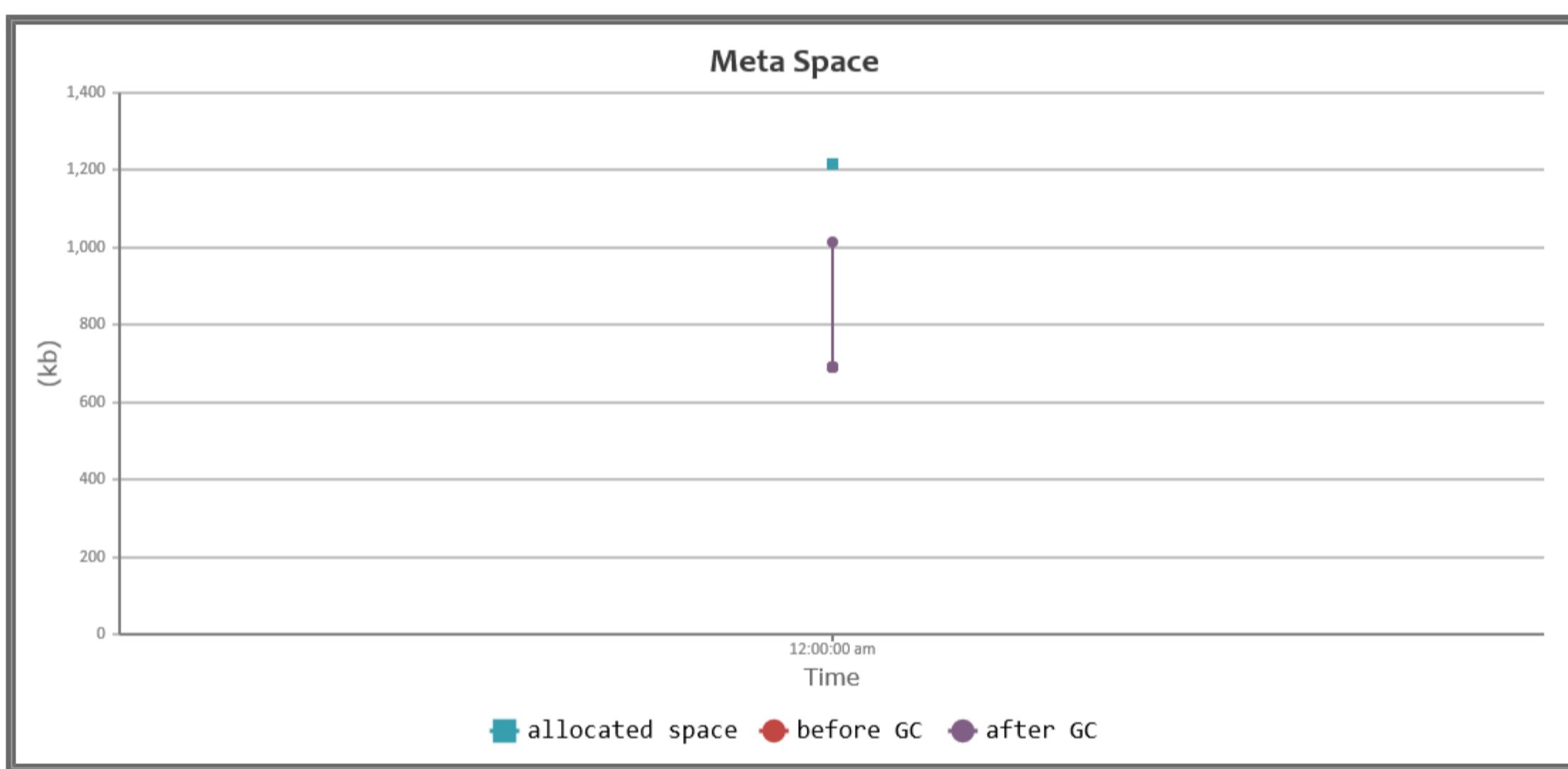
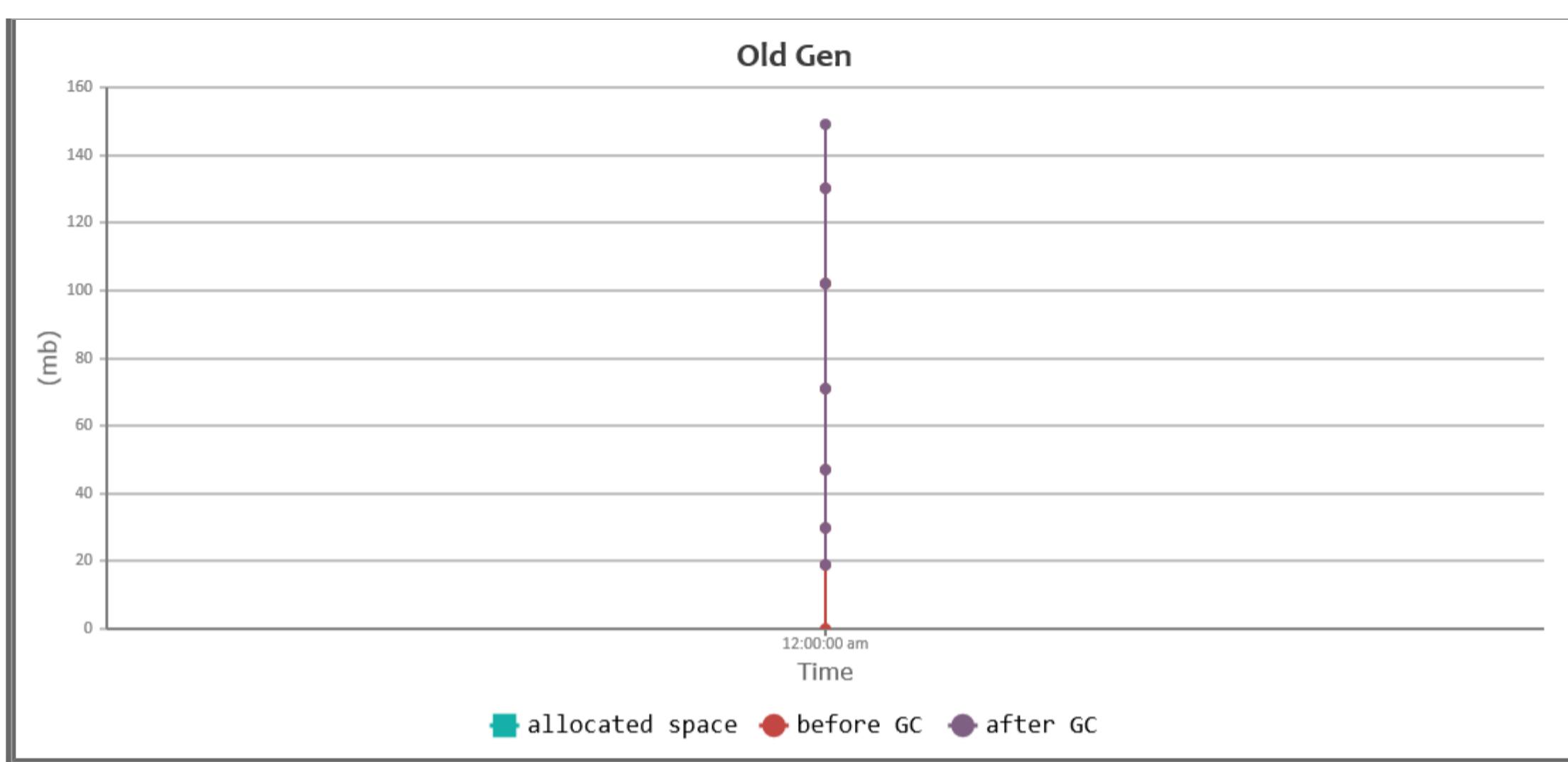


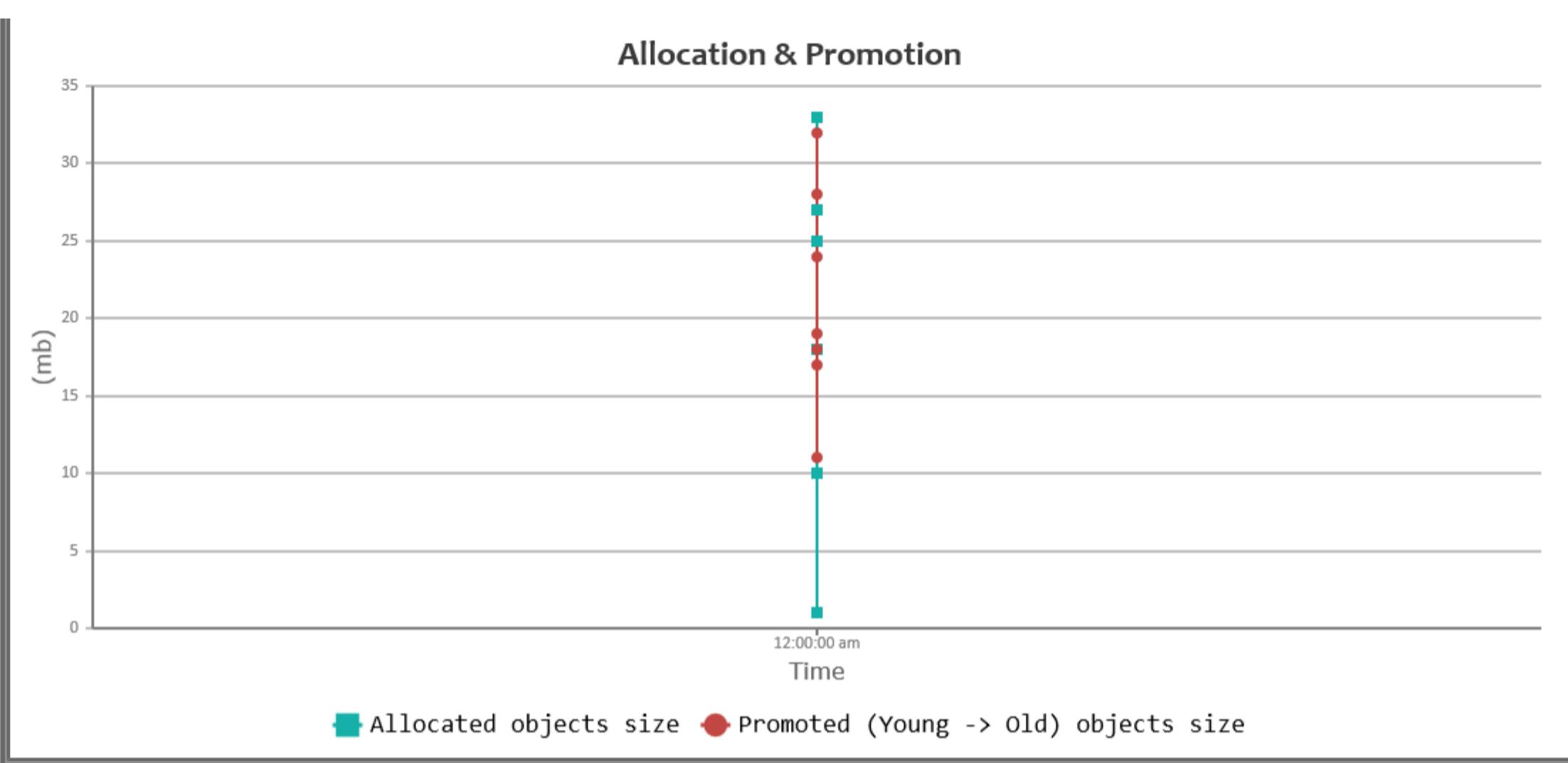
|||| Interactive Graphs (How to zoom graphs?)



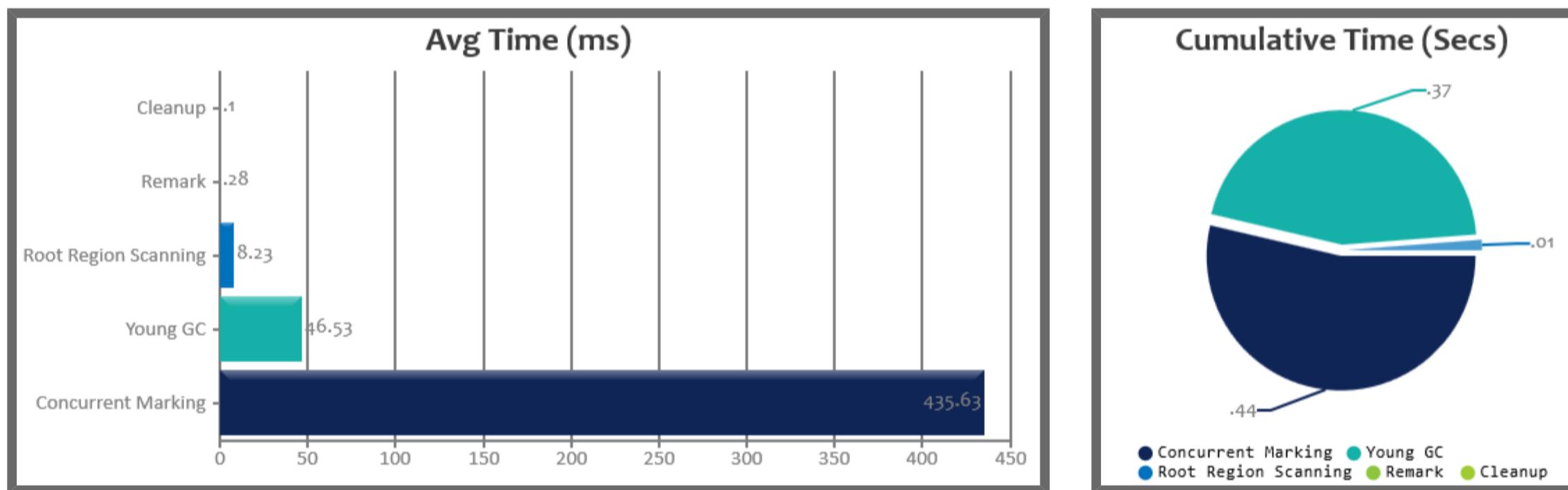






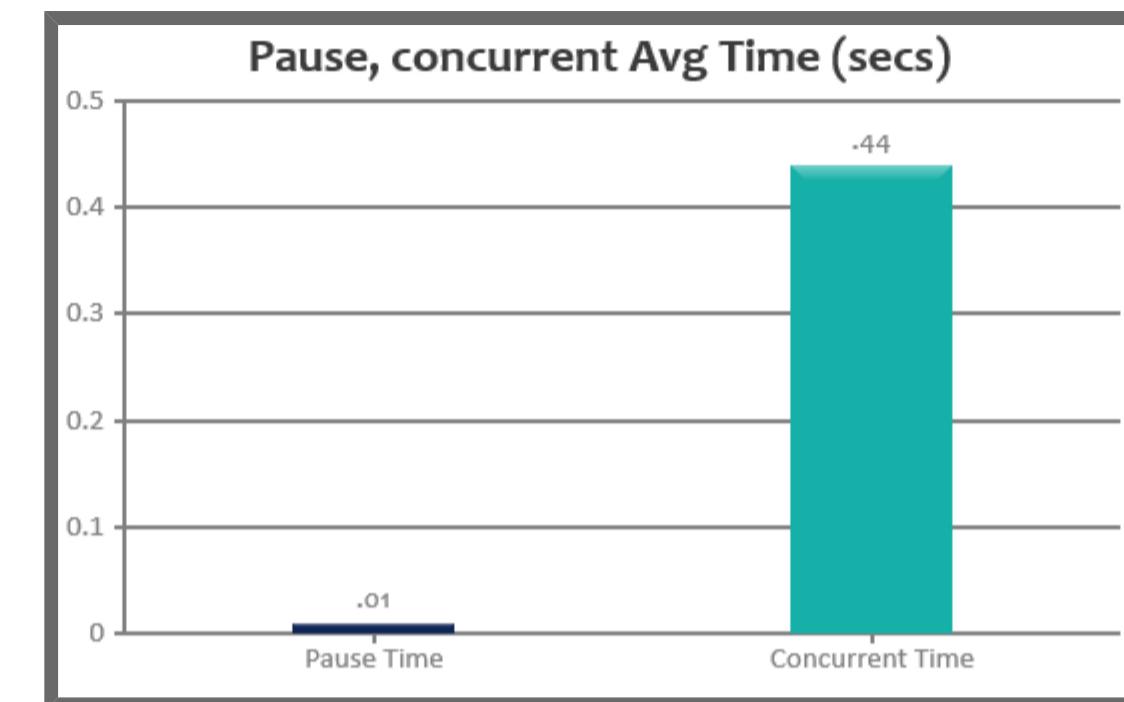
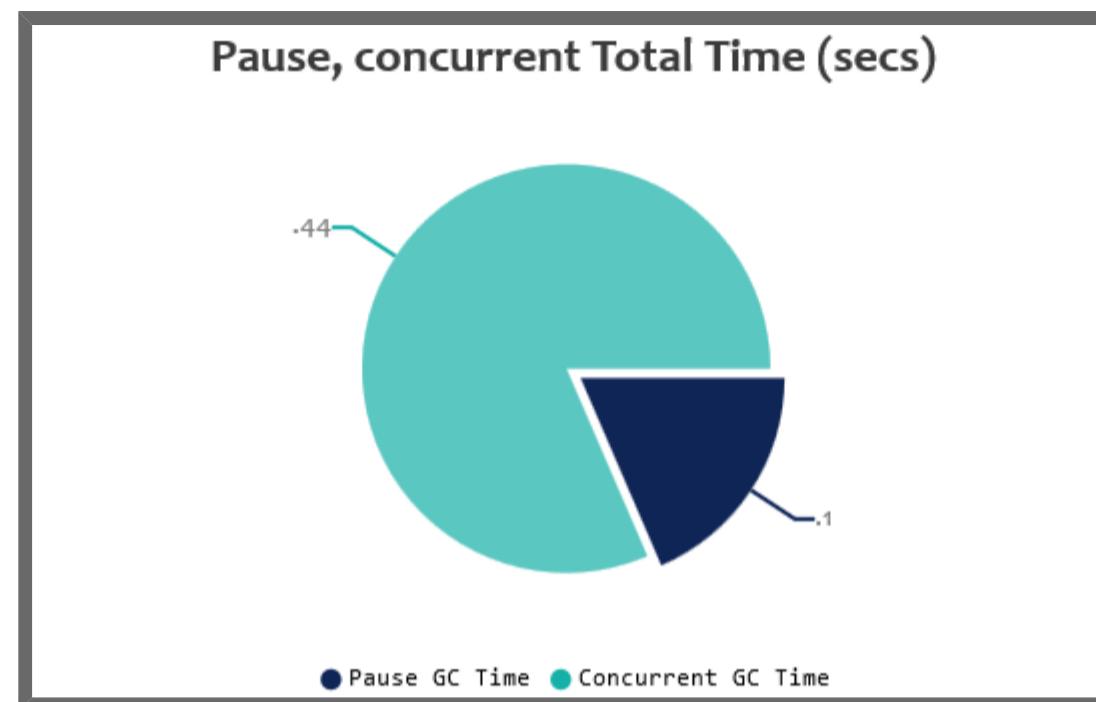


⌚ G1 Collection Phases Statistics



	Concurrent Marking	Young GC ⓘ	Root Region Scanning	Remark ⓘ	Cleanup ⓘ
Total Time ⓘ	436 ms	372 ms	8.23 ms	0.277 ms	0.101 ms
Avg Time ⓘ	436 ms	46.5 ms	8.23 ms	0.277 ms	0.101 ms
Std Dev Time	0	85.4 ms	0	0	0
Min Time ⓘ	436 ms	10.0 ms	8.23 ms	0.277 ms	0.101 ms
Max Time ⓘ	436 ms	272 ms	8.23 ms	0.277 ms	0.101 ms
Interval Time ⓘ	n/a	23.0 ms	n/a	n/a	n/a
Count ⓘ	1	8	1	1	1

⌚ G1 GC Time



Pause Time ?

Total Time	100 ms
Avg Time	11.2 ms
Std Dev Time	7.31 ms
Min Time	0.101 ms
Max Time	20.0 ms

Concurrent Time ?

Total Time	444 ms
Avg Time	444 ms
Std Dev Time	0
Min Time	444 ms
Max Time	444 ms

⚙ Object Stats ?

Total created bytes ?	185 mb
Total promoted bytes ?	149 mb
Avg creation rate ?	34.72 mb/sec
Avg promotion rate ?	27.96 mb/sec

(cpu) CPU Stats ? (To learn more about CPU stats, [click here](#))

CPU Time: ?	660 ms
User Time: ?	580 ms
Sys Time: ?	80.0 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](#))

Not Reported in the log.

⌚ Allocation stall metrics ?

(To learn more about Allocation Stall, [click here](#))

Not Reported in the log.

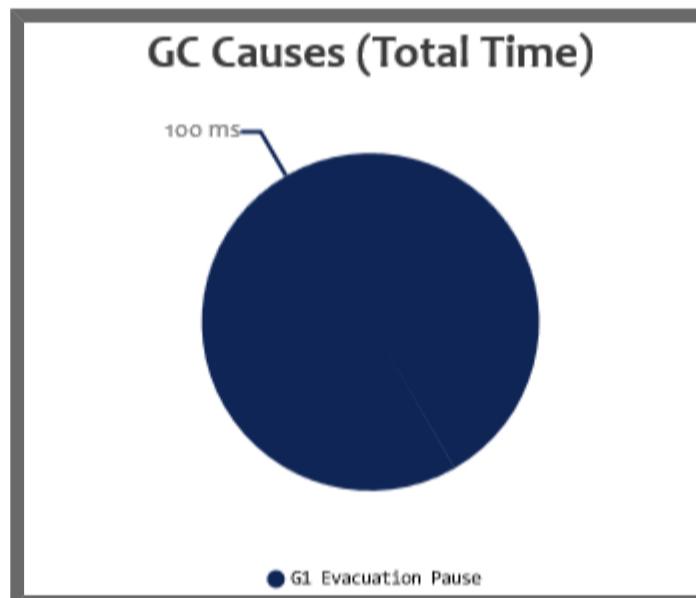
📄 String Deduplication Metrics ?

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 ?	7	14.3 ms	20.0 ms	100 ms



⌚ Tenuring Summary ?

Not reported in the log.

📄 JVM Arguments ?

(To learn about JVM Arguments, [click here](#))

Not reported in the log.