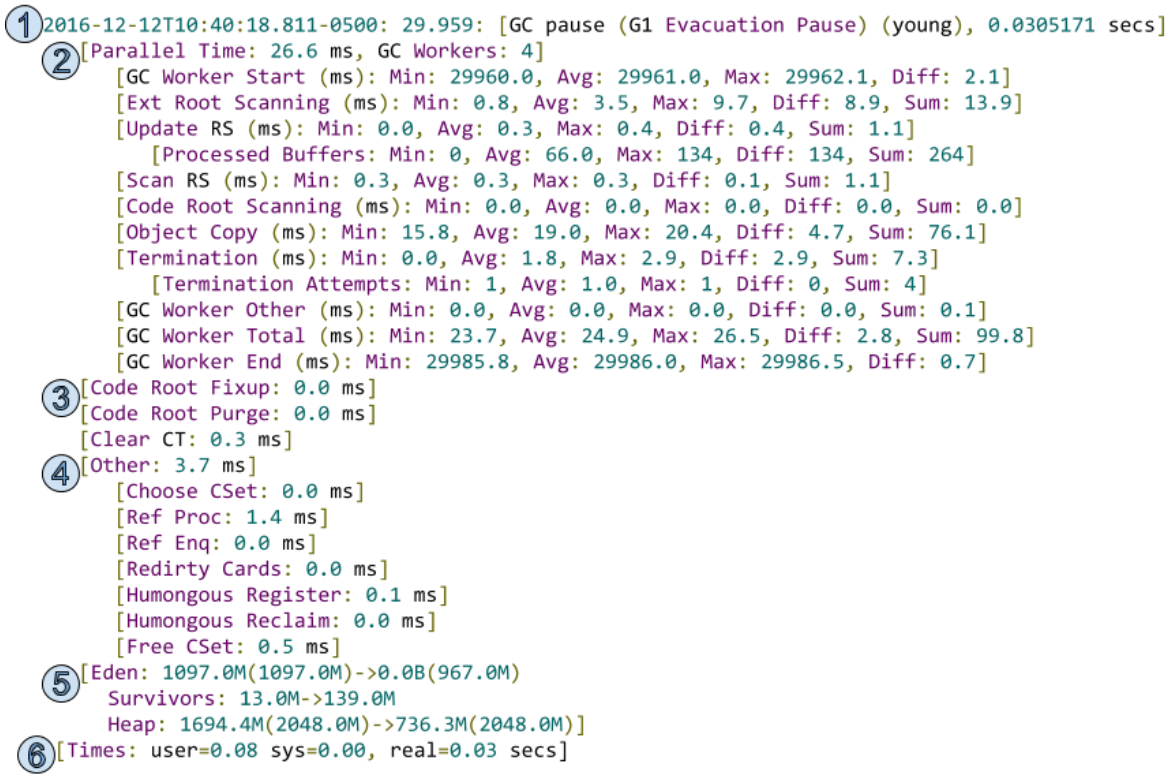

1. **The first point outlines four key pieces of information:**

a. The actual date and time the event occurred, logged by setting '-XX:+PrintGCDateStamps' - **2016-12-12T10:40:18.811-0500**
b. The relative timestamp, since the start of the JVM - **29.959**
c. The type of collection - **G1 Evacuation Pause (young)** - identifies this as an evacuation pause and a young collection

i. The next most popular cause is **G1 Humongous Allocation**

d. The time the collection took - **0.0305171 sec**

2. **The second point outlines all of the parallel tasks:**

a. **Parallel Time** - How much Stop The World (STW) time parallel tasks took, beginning at the start of the collection and finishing when the last GC worker ends - 26.6ms
b. **GC Workers** - The number of parallel GC workers, defined by -XX:ParallelGCThreads - **4**

i. Defaults to the number of CPUs, up to 8. For 8+ CPUs, it defaults to a ⅝ thread to CPU ratio

c. **GC Worker Start** - The min / max timestamp since the start of the JVM when the GC workers began. The difference represents the time in milliseconds

between the start of the first and last thread. Ideally, you want them to start quickly and at the same time

d. **Ext Root Scanning** - The time spent scanning external roots (thread stack roots, JNI, global variables, system dictionary, etc..) to find any that reach into the current collection set
e. **Update RS (Remembered Set or RSet)** - Each region has its own Remembered Set. It tracks the addresses of cards that hold references into a region. As writes occur, a post-write barrier manages changes in inter-region references by marking the newly referred card dirty and placing in on the log buffer or dirty card queue. Once full, concurrent refinement threads process these queues in parallel to running application threads. **Update RS** comes in to enable the GC Workers to process any outstanding buffers which were not handled prior to the start of the collection. This ensures that each RSet is up-to-date

i. **Processed Buffers** - This shows how many Update Buffers were processed during Update RS

f. **Scan RS** - The Remembered Set of each region is scanned to look for references that point to the regions in the Collection Set
g. **Code Root Scanning[**local variables etc.,] - The time spent scanning the roots of compiled source code for references into the Collection Set
h. **Object Copy** - During an evacuation pause, all regions in the Collection Set must be evacuated. Object copy is responsible for copying all remaining live objects to new regions
i. **Termination** - When a GC worker finishes, it enters a termination routine where it synchronizes with the other workers and tries to steal outstanding tasks. The time represents how long it took from when a worker first tried to terminate and when it actually terminated

i. **Termination Attempts** - If a worker successfully steals tasks, it re-enters the termination routine and tries to steal more work or terminate. Every time tasks are stolen and termination is re-entered, the number of termination attempts will be incremented

j. **GC Worker Other** - This represents time spent on tasks not accounted to the previous tasks
k. **GC Worker Total** - This shows the min, max, average, diff and sum for the time spent by each of the parallel worker threads
l. **GC Worker End** - The min / max timestamp since the start of the JVM when the GC workers ended. The diff represents the time in milliseconds between the end of the first and last thread. Ideally, you want them to end quickly and at the same time

3. The third point outlines serial tasks:

a. **Code Root Fixup** - Walking marked methods that point into the CSet to fix any pointers that may have moved during the GC
b. **Code Root Purge** - Purge entries in the code root table
c. **Clear CT** - The card table is cleared of dirtied cards

4. **The fourth point outlines other tasks not previously accounted for. They are also serial.**

a. **Choose CSet** - Selects the regions for the Collection Set
b. **Ref Proc** - Processes any soft/weak/final/phantom/JNI references discovered by the STW reference processor
c. **Ref Enq** - Loops over references and enqueues them to the pending list
d. **Reditry Cards** - Cards modified through the collection process are remarked as dirty
e. **Humongous Register** - With 'G1ReclaimDeadHumongousObjectsAtYoungGC' enabled (default true / feature added in JDK 8u60) G1 will try to eagerly collect Humongous regions during Young GC. This represents how long it took to evaluate if the Humongous regions are candidates for eager reclaim and to record them. Valid candidates will have no existing strong code roots and only sparse entries in the Remembered Set. Each candidate will have its Remembered Set flushed to the dirty card queue and, if emptied, the region will be added to the current Collection Set
f. **Humongous Reclaim** - The time spent ensuring the humongous object is dead and cleaned up, freeing the regions, resetting the region type and returning the regions to the free list and accounting for the freed space
g. **Free CSet** - The now evacuated regions are added back to the free list

5. **The fifth point outlines how the generations have changed and how they have been adapted as a result of the current collection:**

a. **Eden: 1097.0M(1097.0M)->0.0B(967.0M)**

i. This shows that the current Young Collection was triggered because the Eden space was full - 1097.0M of the allocated (1097.0M)
ii. It shows that all Eden regions were evacuated and the usage was reduced to 0.0B by the collection
iii. It also shows that the total allocation of Eden space has been reduced to 967.0M for the next collection

b. **Survivors: 13.0M->139.0M**

i. As a result of the young evacuation, the survivor space grew from 13.0M to 139.0M

c. **Heap: 1694.4M(2048.0M)->736.3M(2048.0M)**

i. At the time of the collection, the overall heap allocation was 1694.4M of the max (2048.0M)
ii. After the collection, the overall heap allocation was reduced to 736.3M and the max heap was unchanged at (2048.0M)

6. **The sixth point represents the time taken for the collection:**

a. **user=0.08**

i. Amount of CPU time spent in the user code within the process during the collection. This accounts for all threads across all CPUs. This does not account for time spent outside the process or time spent waiting. Depending on the number of parallel threads, user time will be quite a bit higher than the real time

b. **sys=0.00**

i. Amount of CPU time spent in the kernel within the process. This accounts for all threads across all CPUs. This does not account for time spent outside the process or time spent waiting

c. **real=0.03**

i. This is the real wall clock time from the start to the end of the collection. This also includes time spent in other process and time spent waiting