Mastering GC: tame the beast and make it your best ally

Jean-Philippe Bempel

@jpbempel

DATADOG

Agenda

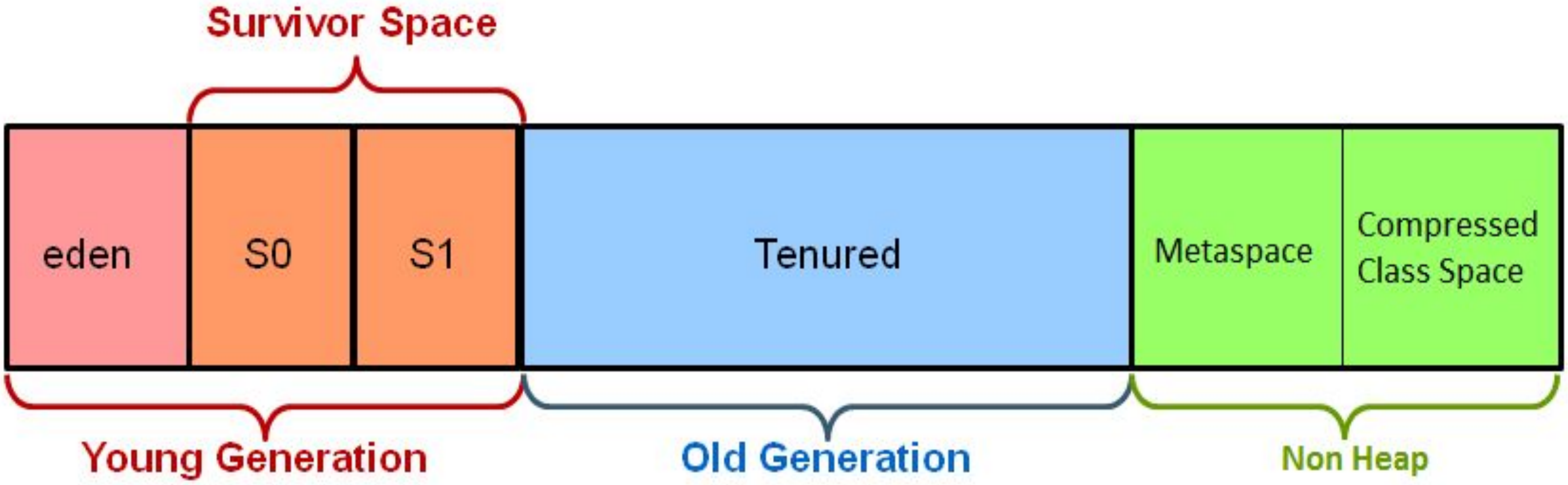
- OpenJDK GCs
  - Serial GC
  - Parallel GC
  - G1
  - Shenandoah
  - Z GC
- Selecting the right GC
- Tuning GC

OpenJDK GCs

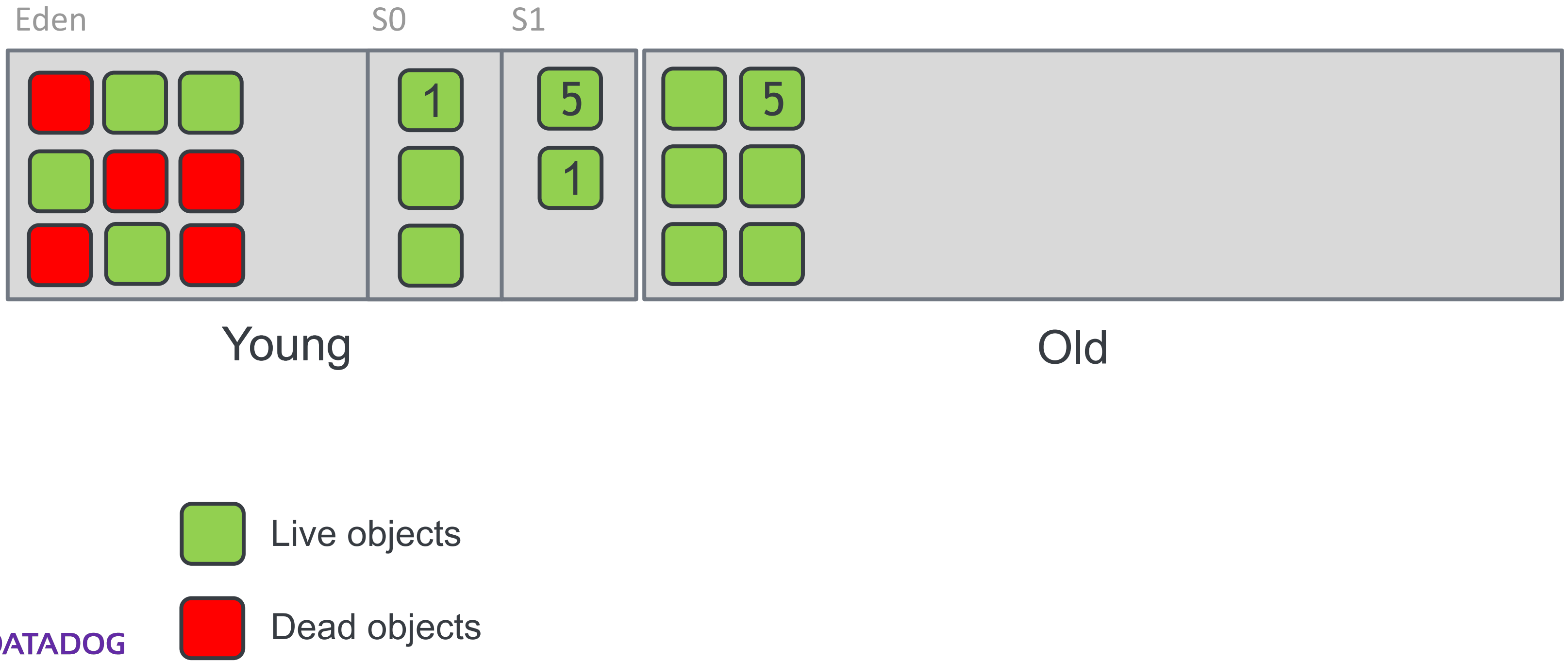
Serial GC

- Generational
- Single threaded
- Stop-The-World
- Default GC for < 2GB or < 2 cores

Spaces & generations



Semi-spaces: Survivors

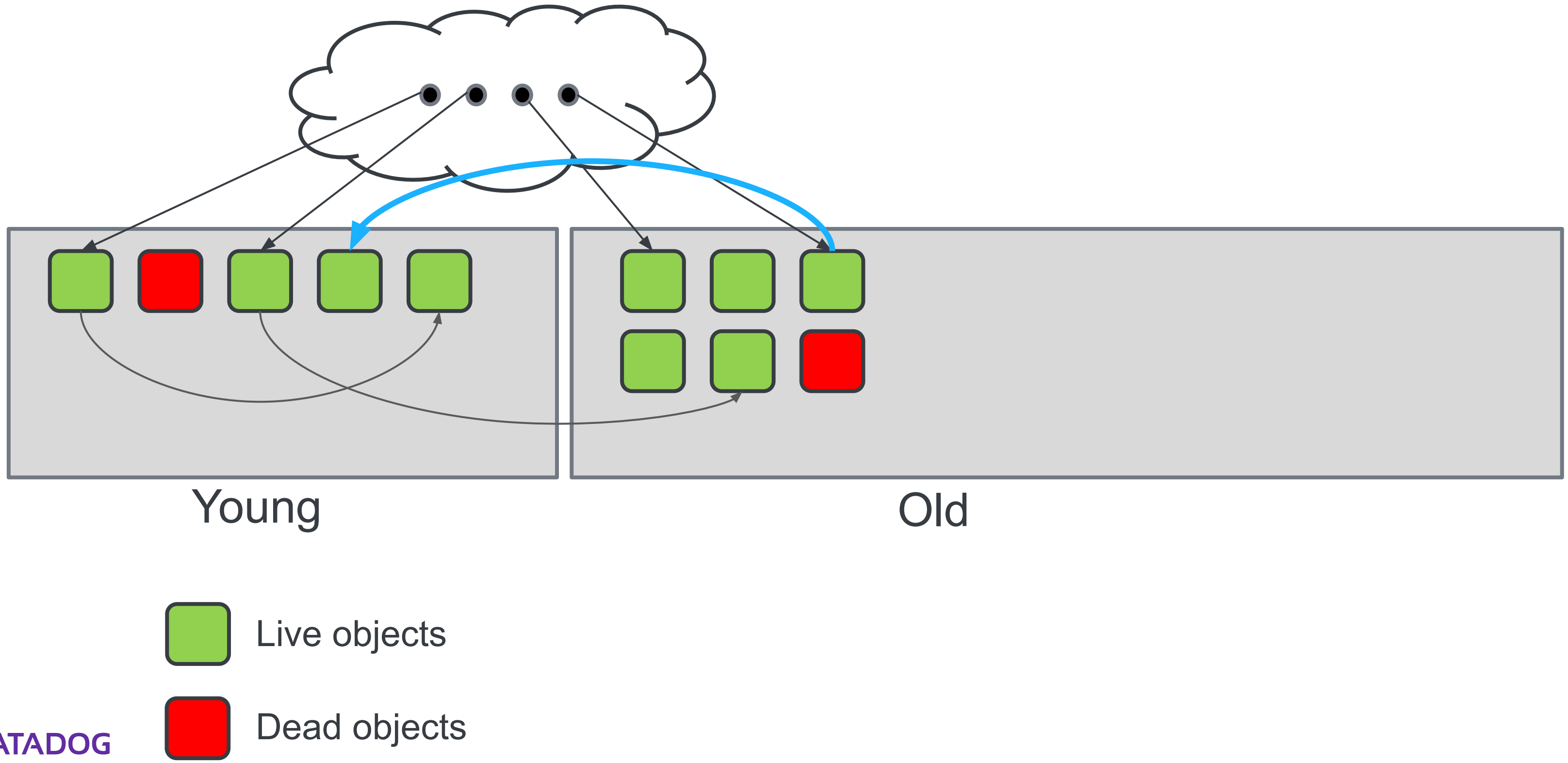


DATADOG

Parallel GC

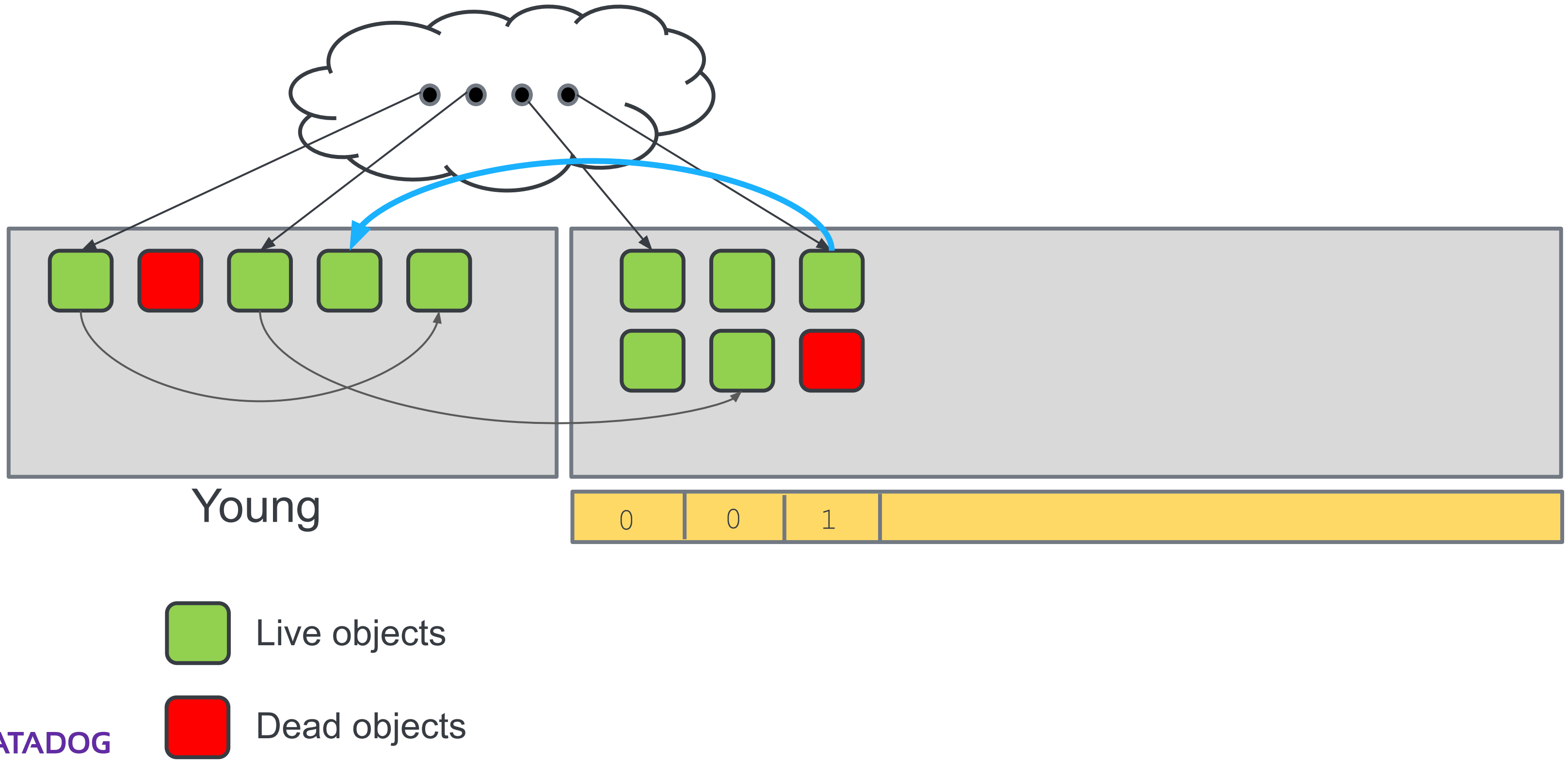
- Generational
- Multi threaded
- Stop-The-World

Minor GC marking



DATADOG

Remember Set: Card Table

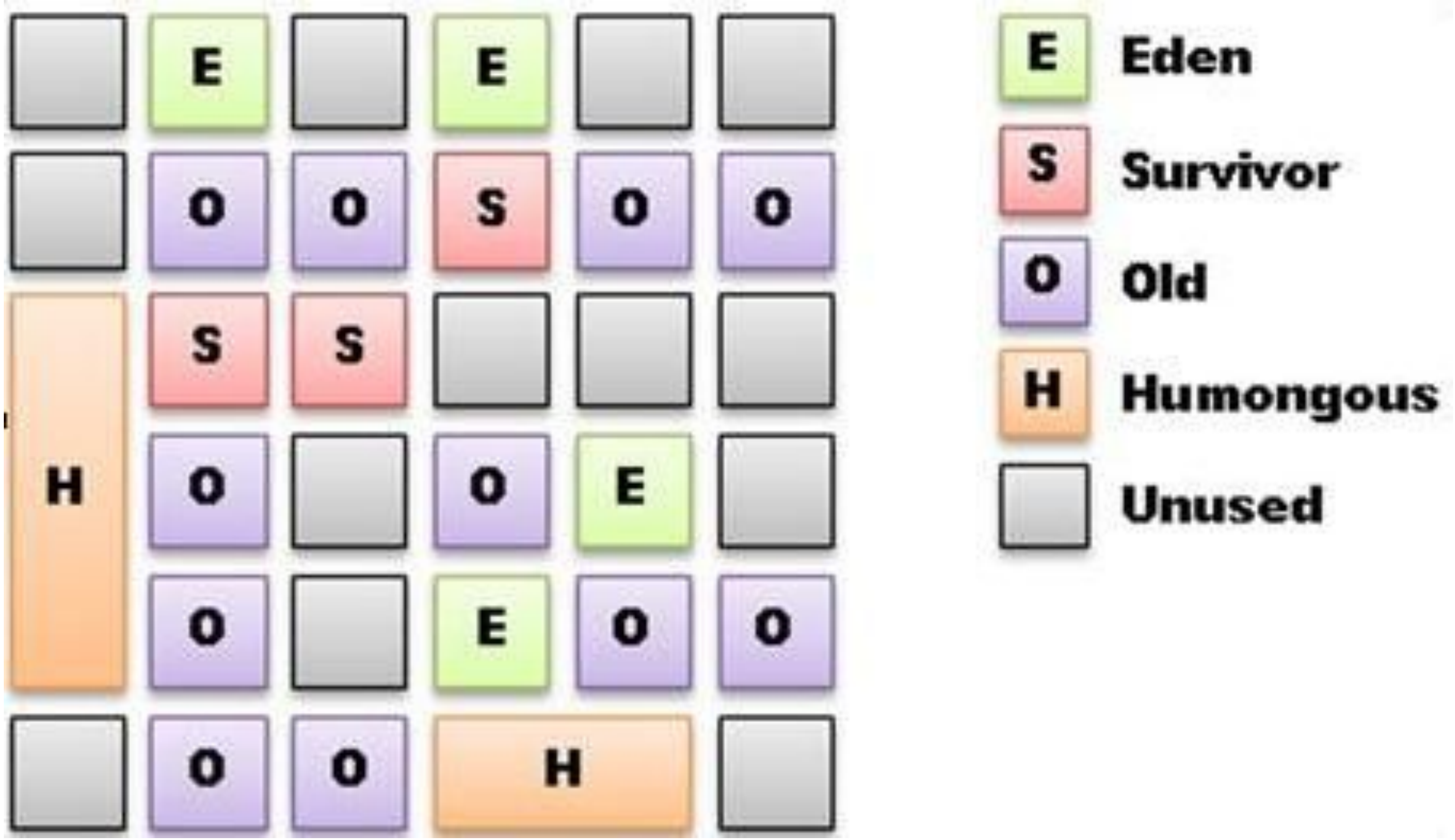


DATADOG

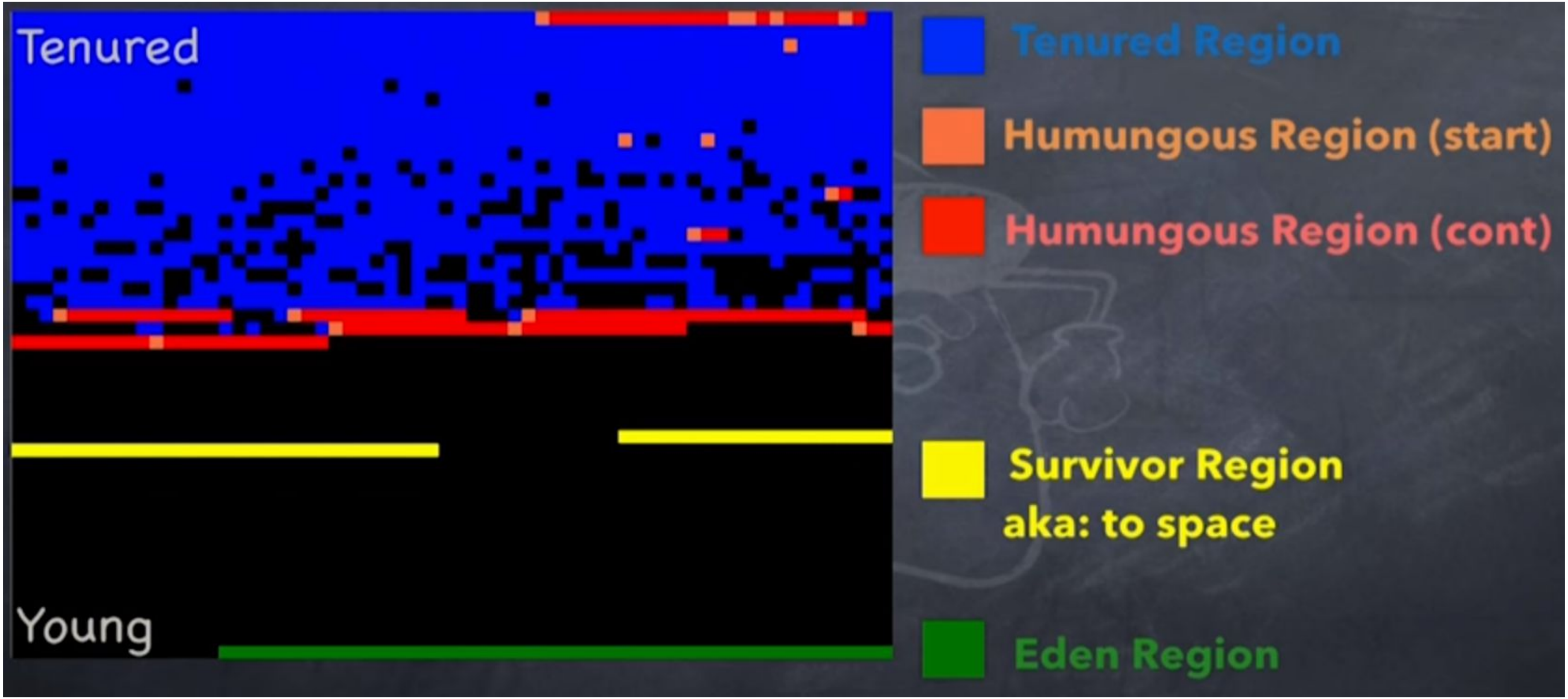
G1 GC

- Generational
- Multi threaded
- Stop-The-World/Concurrent
- Region based

Regions


E
E
O
O
S
O
O
S
S
H
O
O
E
O
E
O
O
O
H
E Eden
S Survivor
O Old
H Humongous
Unused

Regions



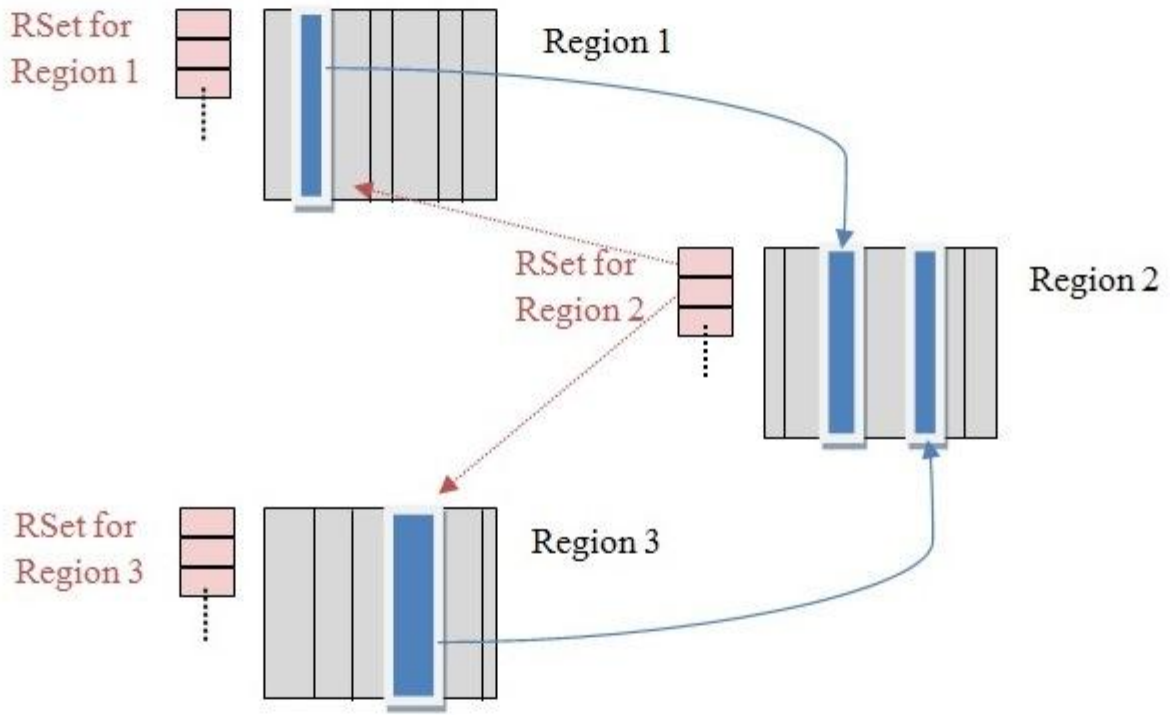
DATADOG

Credit: Kirk Pepperdine

Regions

- Memory reclaimed by evacuation of a region (copying objects/compacting)
- Limiting copy and focus on regions with most garbage => G1
- When evacuating objects, you update refs, but avoid scanning the full heap
- Using RememberSet to know which region to scan for refs

RememberSet



DATADOG

Shenandoah GC

- Not generational (single space)
- Fully Concurrent
- Region based

Concurrent Evacuation: challenges

- How to make sure we can reach an object that is evacuating?
- Read (Load) Barrier
  - Everytime we are loading an object to access it, we check if we need to find it elsewhere or not
- Self-Healing
  - Application thread can help fix addresses not yet fixed by GC threads

Z GC

- Not generational (yet)
- Fully Concurrent
- Region based

Colored pointers

- Store metadata in unused bits of reference address
- 44 bits for addressing 16TB (Max Heap)
- 4 bits for metadata:
  - Marked0
  - Marked1
  - Remapped
  - Finalizable

```
6                 4 4 4  4 4                                             0
3                 7 6 5  2 1                                             0
+-------------------+-+----+-----------------------------------------------+
|00000000 00000000 0|0|1111|11 11111111 11111111 11111111 11111111 11111111|
+-------------------+-+----+-----------------------------------------------+
|                   | |    |
|                   | |    * 41-0 Object Offset (42-bits, 4TB address space)
|                   | |
|                   | * 45-42 Metadata Bits (4-bits)  0001 = Marked0
|                   |                                 0010 = Marked1
|                   |                                 0100 = Remapped
|                   |                                 1000 = Finalizable
|                   |
|                   * 46-46 Unused (1-bit, always zero)
|
* 63-47 Fixed (17-bits, always zero)
```

Selecting the right GC

Identify your workload: Throughput oriented

- Jobs, processing by steps
- High volume of data
- No response to a human user
- Examples: Spark jobs, Kafka consumer/producer, intakes, ETL, …

DATADOG

Identify your workload: Latency sensitive

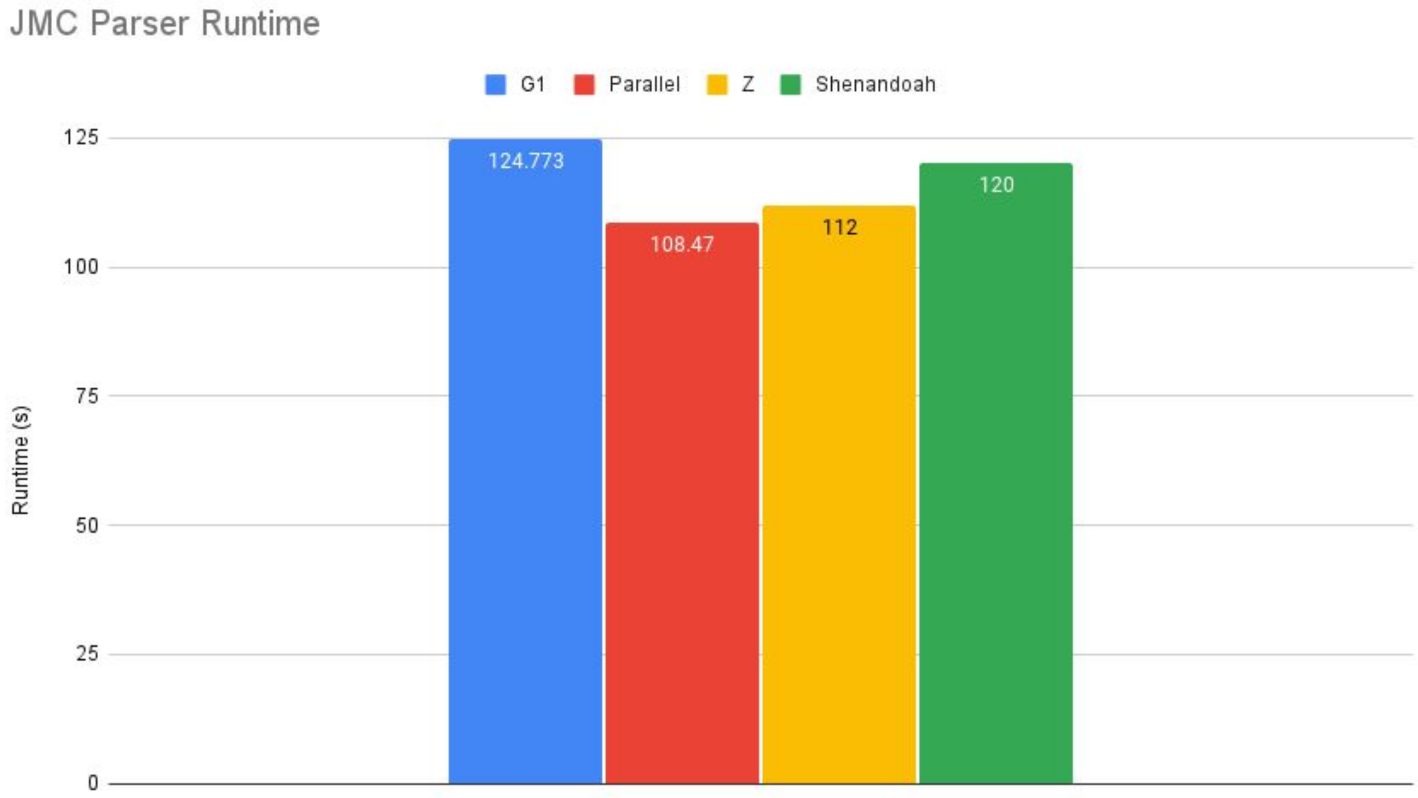
- Application based on Request/Response
- Response to human user, even indirect (microservices dependencies)
- Databases
- Examples: Http/gRPC services, Cassandra

DATADOG

Throughput oriented: Which GC?

- JDK Flight Recorder (JFR) files
- Use JDK Mission Control Library for parsing round robin fashion
- Metric: total runtime of a predefined number of JFR files processed

Throughput oriented: Which GC?



DATADOG

Throughput oriented: Analysis G1

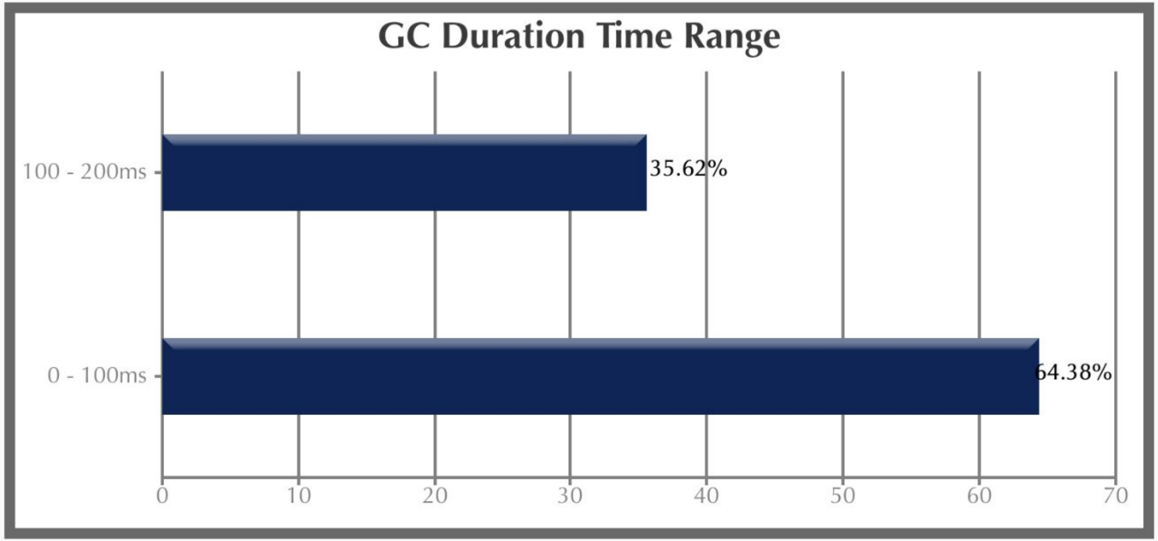
Key Performance Indicators

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

1 Throughput : 85.456%

2 Latency:

Avg Pause GC Time	76.5 ms
Max Pause GC Time	197 ms



GCPauseDuration Time Range :

Duration (ms) 100 ms Change	No. of GCs	Percentage
0 - 100	150	64.38%
100 - 200	83	35.62%

Throughput oriented: Analysis G1

	Young GC	Concurrent Marking	Remark	Cleanup
Total Time	20 sec 478 ms	2 sec 646 ms	41.6 ms	6.02 ms
Avg Time	83.9 ms	56.3 ms	2.31 ms	0.334 ms
Std Dev Time	56.8 ms	80.9 ms	0.243 ms	0.0905 ms
Min Time	1.72 ms	1.72 ms	1.99 ms	0.225 ms
Max Time	326 ms	323 ms	2.91 ms	0.627 ms
Interval Time	501 ms	2 sec 428 ms	6 sec 539 ms	6 sec 539 ms
Count	244	47	18	18

Throughput oriented: Analysis Parallel

Key Performance Indicators

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

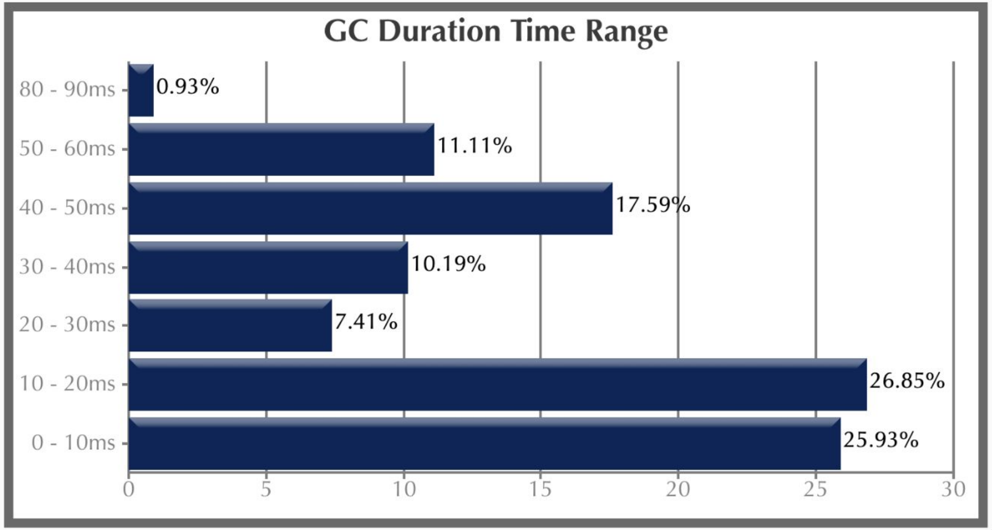
1 Throughput : 97.37%

2 Latency:

Avg Pause GC Time	25.2 ms
Max Pause GC Time	82.0 ms

GCPauseDuration Time Range :

Duration (ms) 10 ms Change	No. of GCs	Percentage
0 - 10	28	25.93%
10 - 20	29	26.85%
20 - 30	8	7.41%
30 - 40	11	10.19%
40 - 50	19	17.59%
50 - 60	12	11.11%
80 - 90	1	0.93%



DATADOG

Throughput oriented: Analysis Parallel

Total GC stats

Total GC count	108
Total reclaimed bytes	161.27 gb
Total GC time	2 sec 717 ms
Avg GC time	25.2 ms
GC avg time std dev	18.7 ms
GC min/max time	1.53 ms / 82.0 ms
GC Interval avg time	949 ms

Minor GC stats

Minor GC count	106
Minor GC reclaimed	157.85 gb
Minor GC total time	2 sec 662 ms
Minor GC avg time	25.1 ms
Minor GC avg time std dev	18.8 ms
Minor GC min/max time	1.53 ms / 82.0 ms
Minor GC Interval avg	967 ms

Full GC stats

Full GC Count	2
Full GC reclaimed	3.42 gb
Full GC total time	55.2 ms
Full GC avg time	27.6 ms
Full GC avg time std dev	11.1 ms
Full GC min/max time	16.5 ms / 38.7 ms
Full GC Interval avg	29 sec 909 ms

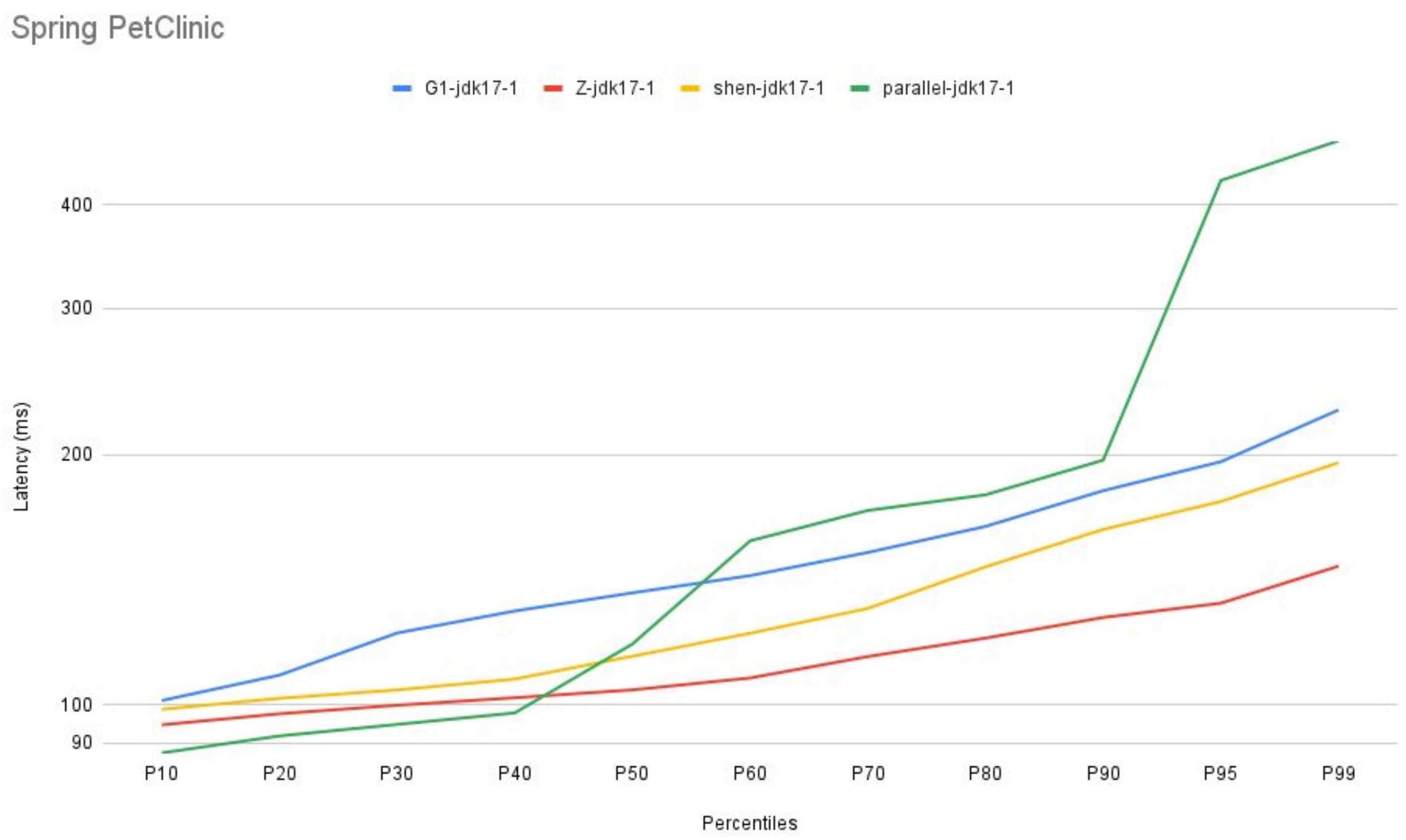
GC Pause Statistics

Pause Count	108
Pause total time	2 sec 717 ms
Pause avg time	25.2 ms
Pause avg time std dev	0.0
Pause min/max time	1.53 ms / 82.0 ms

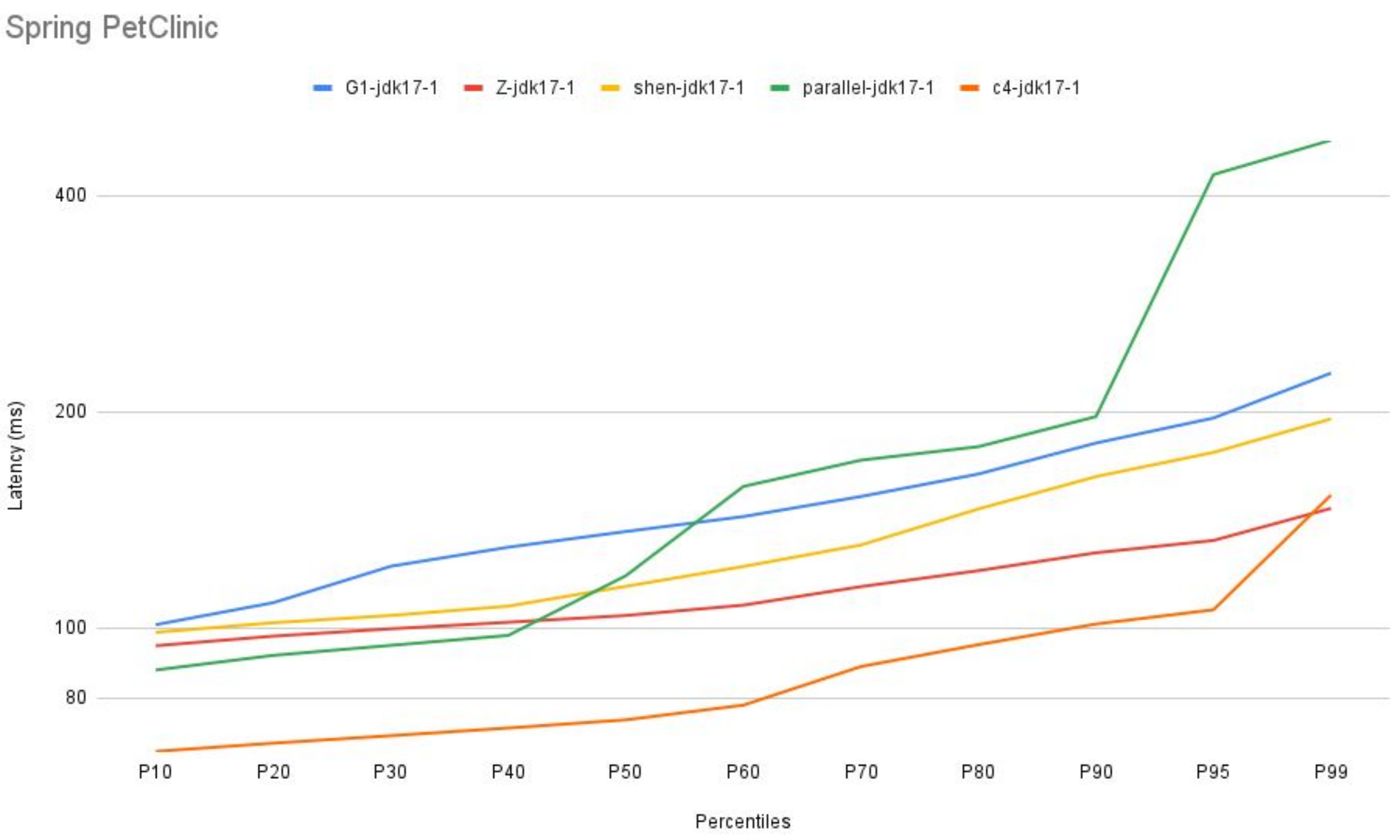
Latency sensitive: Which GC?

- Spring petclinic demo app
- Send requests and waiting for response
- Metric: Percentiles of latency of each request

Latency sensitive: Which GC?



Latency sensitive: Which GC?

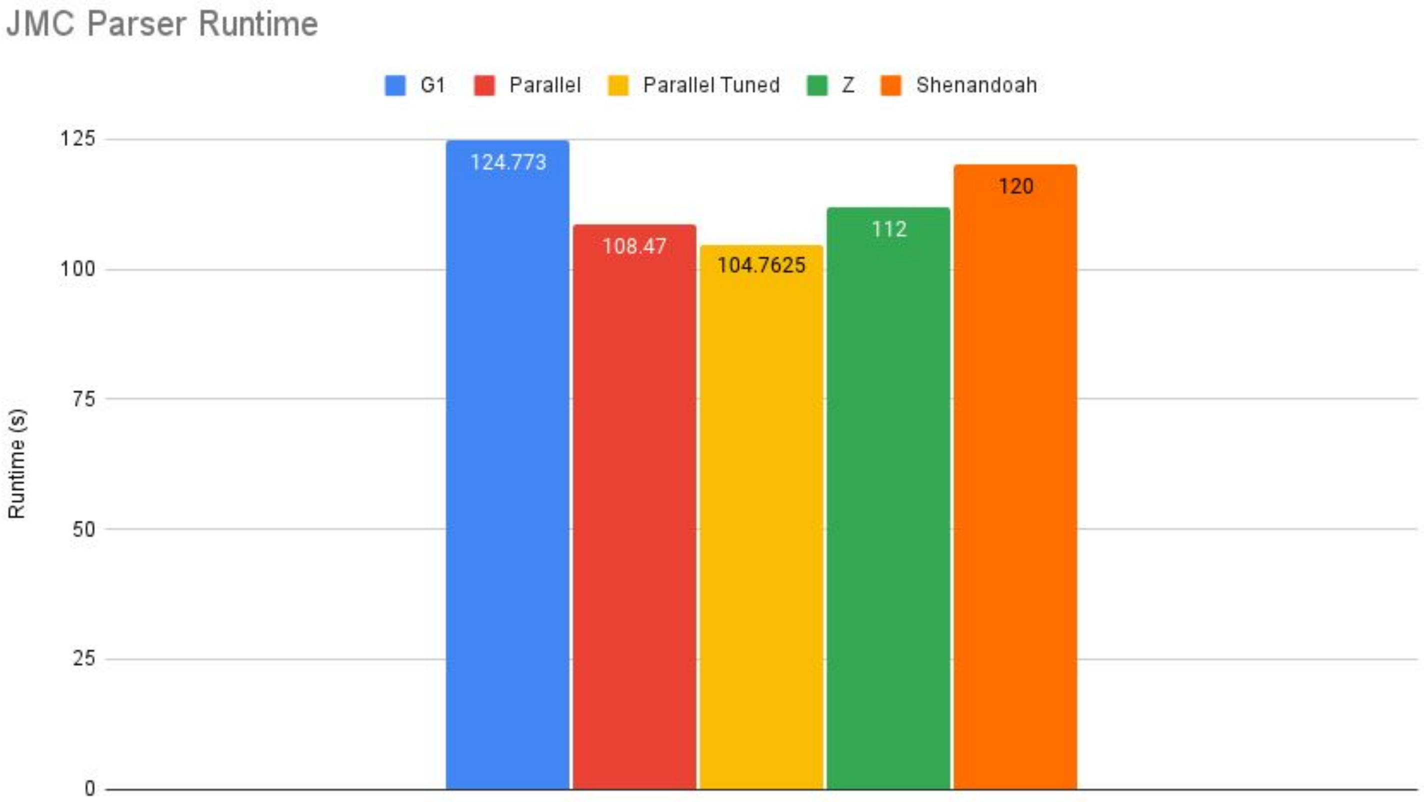


Tuning GC

Tuning Parallel GC

- Very simple to reason about
- Main goal: avoid promoting short/middle lived object to postpone Full GC
- Adjust young gen to reach this goal depending on your workload
- Don’t hesitate to increase the young gen sometimes > 50% of the heap

Tuning Parallel GC

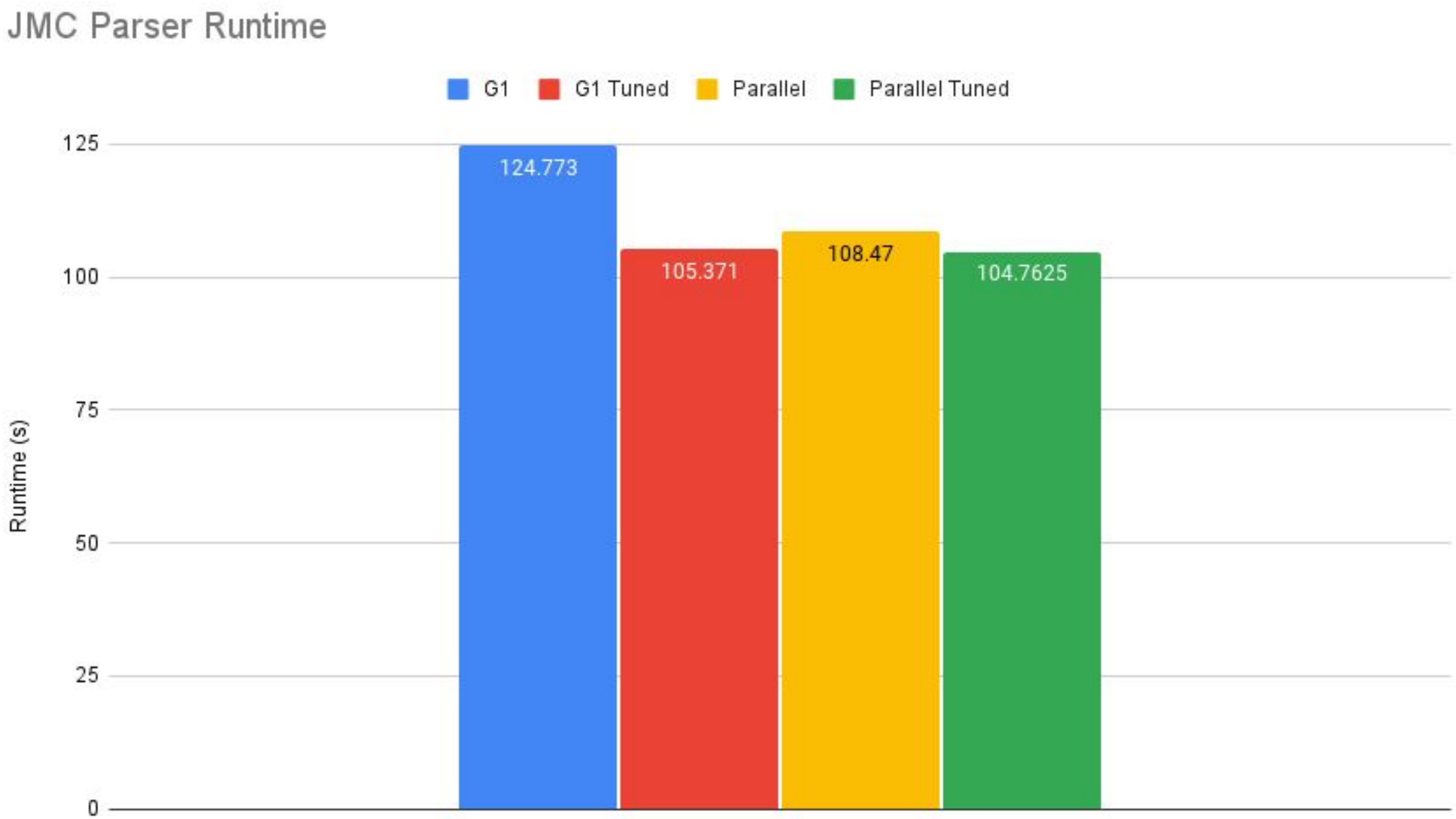


DATADOG

Tuning G1: Humongous

- Humongous objects are difficult to handle for G1:
  - Creates fragmentation
  - Cannot move
  - Triggers prematurely GC b/c checked during each Humongous alloc
- GC Cause = Humongous allocation
  - Try to adjust HeapRegionSize to reduce Humongous objects
  - But large objects not Humongous need to be moved, may increase pause time…
  - Cannot do if regions are already 32MB…

Tuning G1: Humongous



Tuning G1: Young Gen Resize

- Resize based on target pause time:
  - Increase if pause time < target
  - Decrease if pause time > target

- Goal overshoot leads to aggressive young gen reduction down to 5%:
  - Short young gen generates very frequent Young GC
  - This storm can lead to over promotion/copy of objects and increase of pause time
  - More frequent GCs + significant pause time => worse than overshooting the initial goal

- Use NewSize/MaxNewSize for min and max Young Gen
  - But not NewRatio or Xmn which will fix the Young Gen

Tuning G1: Target pause time

- Main knob for G1 to tune is MaxGCPauseMillis as Target Pause time
- Increase target pause for more throughput (accepting more pause time)
- Decrease target pause time for more low latency, but below 50ms it's difficult…

Tuning Shenandoah

- To be effective, allocation rate should not overrun the GC
- Otherwise => Pacer or Full GC
- Need heap headroom for having time to collect and reclaim regions
- cores/Conc Gc threads for finishing cycle faster
- Region reclaim is done at the end of the GC cycle

Tuning Z

- To be effective, allocation rate should not overrun the GC
- Otherwise “Allocation Stall” but No Full GC
- Need heap headroom for having time to collect and reclaim regions
- cores/Conc Gc threads for finishing cycle faster
- Region reclaim is done as soon as relocation is done

Conclusion

How to choose a GC?

- Throughput oriented workload
  - Parallel GC
  - G1 if no issue and good figures

- Latency sensitive workload
  - Shenandoah
  - Z
  - C4

Why not G1?

- Complexity of the heuristic make it difficult to tune

- Couple of things can go wrong:
    - Fragmentation leading to FullGC, not all Old region are considered
    - RememberSet granularity, Post-Write Barrier, Refinement Threads
    - Evacuation failure (=> full GC), InitiatingHeapOccupancyPercent, G1ReservePercent
    - Young Gen dynamic sizing, drastic reduction leads to minor GC storm
    - Humongous allocations, premature minor GC and fragmentation

- G1 Heuristic can save you or can curse you!

References

References

Understanding low latency GCs
G1 One Garbage Collector to rule them all
Tips for Tuning The G1 GC
G1 Garbage Collector Details and Tuning
What’s the deal with humongous objects in Java?
Shenandoah: The Garbage Collector That Could
Load Reference Barriers
Eliminating forward pointer word
Concurrent GC collectors: ZGC & Shenandoah
Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK
GCEasy.io

Thank You!

Jean-Philippe Bempel

@jpbempel

DATADOG