Parameters to consider for GC algorithms

* HeapSize
* Application DataSetSize
* Number of cpus
* Pausetime : The pause time is the duration during which the garbage collector stops the application to reclaim memory.
* ThroughPut: By this, we mean the time processes spend actually doing application work. **The higher the application time vs. overhead time spent in doing GC work, the higher the throughput of the application**.
* MemoryFootPrint :
* Promptness: This is the time between when an object becomes dead and when the memory it occupies is reclaimed.larger heapsize larger promptness
* Latency: This is the responsiveness of an application. GC pauses affect this variable directly.

**SerialGc:**

The serial collector uses a single thread to perform all the garbage collection work. It's selected by default on certain small hardware and operating system configurations, or it can be explicitly enabled with the option *-XX:+UseSerialGC*.

Pros:

* Without inter-thread communication overhead, it's relatively efficient.
* It's suitable for client-class machines and embedded systems.
* It's suitable for applications with small datasets.
* Even on multiprocessor hardware, if data sets are small (up to 100 MB), it can still be the most efficient.

Cons:

* It's not efficient for applications with large datasets.
* It can't take advantage of multiprocessor hardware.

**Parallel/Throughput *GC java8 and before***

This collector **uses multiple threads to speed up garbage collection**. In Java version 8 and earlier, it's the default for server-class machines. We can override this default by using the -XX:+UseParallelGC option.

Pros:

* It can take advantage of multiprocessor hardware.
* It's more efficient for larger data sets than serial GC.
* It provides high overall throughput.
* It attempts to minimize the memory footprint.

Cons:

* Applications incur **long pause times during stop-the-world operations**.
* It doesn't scale well with heap size.

It's best if we want more throughput and don't care about pause time, as is the case with non-interactive apps like batch tasks, offline jobs, and web servers.

### ****. Concurrent Mark Sweep (CMS) GC****

We consider CMS a mostly concurrent collector. This means **it performs some expensive work concurrently with the application**. It's designed for low latency by eliminating the long pause associated with the full GC of parallel and serial collectors.

We can use the option *-XX:+UseConcMarkSweepGC* to enable the CMS collector. The core Java team deprecated it as of Java 9 and completely removed it in Java 14.

Pros:

* It's great for low latency applications as it minimizes pause time.
* It scales relatively well with heap size.
* It can take advantage of multiprocessor machines.

Cons:

* It's deprecated as of Java 9 and removed in Java 14.
* It becomes relatively inefficient when data sets reach gigantic sizes or when collecting humongous heaps.
* It requires the application to share resources with GC during concurrent phases.
* There may be throughput issues as there's more time spent overall in GC operations.
* Overall, it uses more CPU time due to its mostly concurrent nature.

### ****G1 (Garbage-First) GC java9,11****

G1 uses multiple background GC threads to scan and clear the heap just like CMS. Actually, the core Java team designed G1 as an improvement over CMS, patching some of its weaknesses with additional strategies.

In addition to the incremental and concurrent collection, **it tracks previous application behavior and GC pauses to achieve predictability**. It then focuses on reclaiming space in the most efficient areas first — those mostly filled with garbage. We call it Garbage-First for this reason.

Since Java 9, G1 is the default collector for server-class machines. We can explicitly enable it by providing *-XX:+UseG1GC* on the command line.

Pros:

* It's very efficient with gigantic datasets.
* It takes full advantage of multiprocessor machines.
* It's the most efficient in achieving pause time goals.

Cons:

* It's not the best when there are strict throughput goals.
* It requires the application to share resources with GC during concurrent collections.

G1 works best for applications with very strict pause-time goals and a modest overall throughput, such as real-time applications like trading platforms or interactive graphics programs.

### 4.5. Z Garbage Collector (ZGC)

ZGC is a scalable low latency garbage collector. It manages to keep low pause times on even multi-terabyte heaps. It uses techniques including reference coloring, relocation, load barriers and remapping. It is a good fit for server applications, where large heaps are common and fast application response times are required.

It was introduced in Java 11 as an experimental GC implementation. We can explicitly enable it by providing -XX:+UnlockExperimentalVMOptions -XX:+UseZGC on the command line.