https://www.maknesium.de/21-most-important-java-8-vm-options-for-servers

http://www.informit.com/articles/article.aspx?p=2496621&seqNum=4

https://www.cakesolutions.net/teamblogs/low-pause-gc-on-the-jvm

https://blog.takipi.com/garbage-collectors-serial-vs-parallel-vs-cms-vs-the-g1-and-whats-new-in-java-8/

http://java.sys-con.com/node/4003696

-Xms4g -Xmx4g -XX:PermSize=768m -XX:MaxPermSize=768m

GC\_OPTIONS="-XX:+UseParNewGC \  
    -XX:+UseConcMarkSweepGC \  
    -XX:+PrintGCDateStamps \  
    -XX:+PrintGCDetails \  
    -XX:SurvivorRatio=6 \  
    -XX:CMSFullGCsBeforeCompaction=1 \  
    -XX:TargetSurvivorRatio=90 \  
    -XX:CMSInitiatingOccupancyFraction=75 \  
    -XX:+UseCMSInitiatingOccupancyOnly \  
    -XX:+CMSClassUnloadingEnabled  \  
    -XX:+CMSParallelRemarkEnabled \  
    -XX:+DisableExplicitGC \  
    -XX:ParallelGCThreads=32 \  
    -XX:ParallelCMSThreads=5 \  
    -verbose:class \  
    -verbose:gc"

-Xms532m -Xmx532m -XX:MetaspaceSize=532m -XX:MaxMetaspaceSize=532m -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+PrintGCDateStamps -XX:+PrintGCDetails -XX:ParallelGCThreads=12 -XX:q=85 -XX:+UseCMSInitiatingOccupancyOnly -XX:SoftRefLRUPolicyMSPerMB=100 -XX:NewRatio=3 -XX:+CMSParallelRemarkEnabled -XX:+DisableExplicitGC -XX:+HeapDumpOnOutOfMemoryError -XX:ParallelGCThreads=8 -XX:ParallelCMSThreads=4 -verbose:gc

-Xms8192m -Xmx8192m -XX:PermSize=768m -XX:MaxPermSize=1500m -XX:NewRatio=3  -verbose:gc -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+PrintGCDateStamps -XX:+PrintGCDetails  -XX:SurvivorRatio=6 -XX:+UseCMSInitiatingOccupancyOnly -XX:+CMSParallelRemarkEnabled -XX:+DisableExplicitGC -XX:+HeapDumpOnOutOfMemoryError -XX:ParallelGCThreads=32 -XX:ParallelCMSThreads=5

1. When using -XX:TargetSurvivorRatio=90 will this leave ten percent of to-space for objects to be moved from eden?

No. It means that a tenuring threshold is chosen so that, based on the ages of what was scavenged in the last minor collection, there should be nearly 90% of the survivor size used. The actual amount scavenged from either the survivor space or eden may be considerably more or less.

1. What is the Parallel Garbage collector (-XX:+UseParallelGC)?

The new parallel garbage collector is similar to the young generation collector in the default garbage collector but uses multiple threads to do the collection. By default on a host with N CPUs, the parallel garbage collector uses N garbage collector threads in the collection. The number of garbage collector threads can be controlled with a command line option (see below). On a host with a single CPU the default garbage collector is used even if the parallel garbage collector has been requested. On a host with 2 cpus the Parallel garbage collector generally performs as well as the default garbage collector and a reduction in the young generation garbage collector pause times can be expected on hosts with more than 2 cpus.

This new parallel garbage collector can be enabled by using command line product flag -XX:+UseParallelGC. The number of garbage collector threads can be controlled with the ParallelGCThreads command line option (-XX:ParallelGCThreads=<desired number>). This collector cannot be used with concurrent low pause collector.

1. What is the Parallel Young Generation collector (-XX:+UseParNewGC)?

The parallel young generation collector is similar to the parallel garbage collector (-XX:+UseParallelGC) in intent and differs in implementation. Most of the above description for the parallel garbage collector (-XX:+UseParallelGC) therefore applies equally for the parallel young generation collector. Unlike the parallel garbage collector (-XX:+UseParallelGC) this parallel young generation collector can be used with the concurrent low pause collector that collects the tenured generation.

1. What are the phases of the concurrent low pause collector?

There are six phases involved in the collection:

Phase 1 (Initial Checkpoint) involves stopping all the Java threads, marking all the objects directly reachable from the roots, and restarting the Java threads.

Phase 2 (Concurrent Marking) starts scanning from marked objects and transitively marks all objects reachable from the roots. The mutators are executing during the concurrent phases 2, 3, and 5 below and any objects allocated in the CMS generation during these phases (including promoted objects) are immediately marked as live.

Phase 3: During the concurrent marking phase mutators may be modifying objects. Any object that has been modified since the start of the concurrent marking phase (and which was not subsequently scanned during that phase) must be rescanned. Phase 3 (Concurrent Precleaning) scans objects that have been modified concurrently. Due to continuing mutator activity the scanning for modified cards may be done multiple times.

Phase 4 (Final Checkpoint) is a stop-the-world phase. With mutators stopped the final marking is done by scanning objects reachable from the roots and by scanning any modified objects. Note that after this phase there may be objects that have been marked but are no longer live. Such objects will survive the current collection but will be collected on the next collection.

Phase 5 (Concurrent Sweep) collects dead objects. The collection of a dead object adds the space for the object to a free list for later allocation. Coalescing of dead objects may occur at this point. Note that live objects are not moved.

Phase 6 (Resetting) clears data structures in preparation for the next collection.

1. Why is fragmentation a potential problem for the concurrent low pause collector?

Normally the concurrent low pause collector does not copy nor compact the live objects. A garbage collection is done without moving the live objects. If fragmentation becomes a problem, allocate a larger heap. In 1.4.2 if fragmentation in the tenured generation becomes a problem, a compaction of the tenured generation will be done although not concurrently. In 1.4.1 that compaction will occur if the UseCMSCompactAtFullCollection option is turned on.

-XX:+UseCMSCompactAtFullCollection

1. What is the Concurrent Mark Sweep (CMS) collector?

The Concurrent Mark Sweep (CMS) collector (also referred to as the concurrent low pause collector) collects the tenured generation. It attempts to minimize the pauses due to garbage collection by doing most of the garbage collection work concurrently with the application threads.

1. What should I do if my application has mid- or long-lived objects?

Objects that survive a young generation collection have a copying cost (part of the algorithm for a young generation collection is to copy any objects that survive). Mid- or long-lived objects may be copied multiple times. Use the -XX option MaxTenuringThreshold to determine the copying costs. Use -XX:MaxTenuringThreshold=0 to move an object that survives a young generation collection immediately to the tenured generation. If that improves the performance of the application, the copying of long-lived objects is significant. Note that the throughput collector does not use the MaxTenuringThreshold parameter.

1. When is a garbage collection started?

In the default garbage collector a generation is collected when it is full (i.e., when no further allocations can be done from that generation). This is also true of the throughput collector. The concurrent low pause collector starts a collection when the occupancy of the tenured generation reaches a specified value (by default 68%). The incremental low pause collector collects a portion of the tenured generation during each young generation collection. A collection can also be started explicitly by the application.

1. What is the best size for the young generation?

The young generation should be sized large enough so that short-lived objects have a chance to die before the next young generation collection. This is a tradeoff since a larger young generation will allow more time for objects to die but may also take longer to collect. Experiment with the size of the young generation to optimize the young generation collection time or the application throughput.

1. Between NewSize and NewRatio which option takes precedence?

In jdk 1.4.1 and later, neither has strict precedence. The maximum of NewSize and the size calculated using NewRatio is used. The formula is

min(MaxNewSize, max(NewSize, heap/(NewRatio+1)))