Garbage Collection Schemes

Sang Shin
Michèle Garoche
http://www.javapassion.com
“Learn with Passion!”

Agenda

- Why you care on GC (as a developer)?
- What is and Why Generational GC?
- GC Performance Metrics
- GC Algorithms
- Types of Collectors
  - Serial collector
  - Parallel collector
  - Parallel compact collector
  - Concurrent Mark-Sweep (CMS) collector: regular, incremental
- Ergonomics

Why you care on GC?

- In general, the default setting should work fine for most applications
- For GC sensitive applications, however, choosing a wrong GC scheme could result in a less than desirable performance

What is and Why Generational GC?

Issues of Non-Generational GC

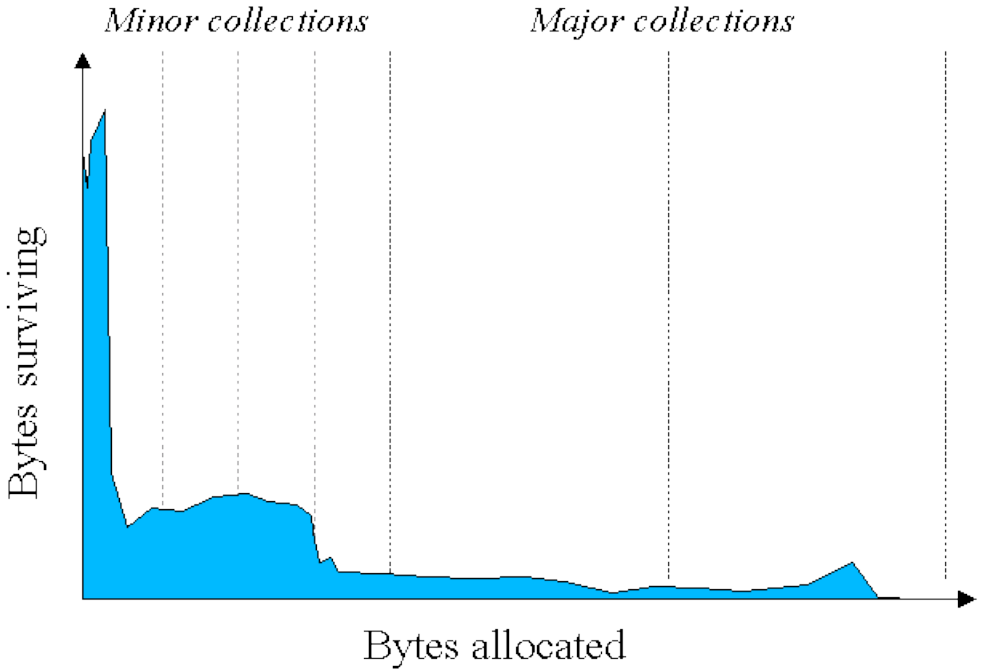
- Most straightforward GC will just iterate over every object in the heap and determine if any other objects reference it
  - This is the behavior of Non-generational GC
  - This gets really slow as the number of objects in the heap increases
  - This does not take advantage of the memory usage behavior of typical Java objects
- Hence the reason for Generational GC

Memory Usage Behavior of Typical Java Objects

- Typical Java object (Young object) is most likely to die shortly after it was created
  - It is called “high infant mortality”
  - Example: Local objects
- Objects that have been around for a while (Old objects) will likely stay around for a while
  - Example: Objects initialized at the time of application startup
- Only a few references from old objects to young objects exist

Empirical Statistics

- Most objects are very short lived
  - 80-98% of all newly allocated objects die shortly after they are created
  - 80-98% of all newly allocated objects die before another megabyte has been allocated



Heap Space With Generations

- Heap space is organized into “generations”
  - Young generation (for young objects)
  - Old (or Tenured) generation (for old objects)
  - Perm generation (meta data, classes, etc.)

Principles of Generational GC

- Keep young and old objects separate
  - Mixing them in a single space would result in inefficient GC
- Use different GC algorithms for different generation heap spaces
  - Objects in each generation have different life-cycle tendencies (memory usage behavior)
  - “Use the right tool for the job”

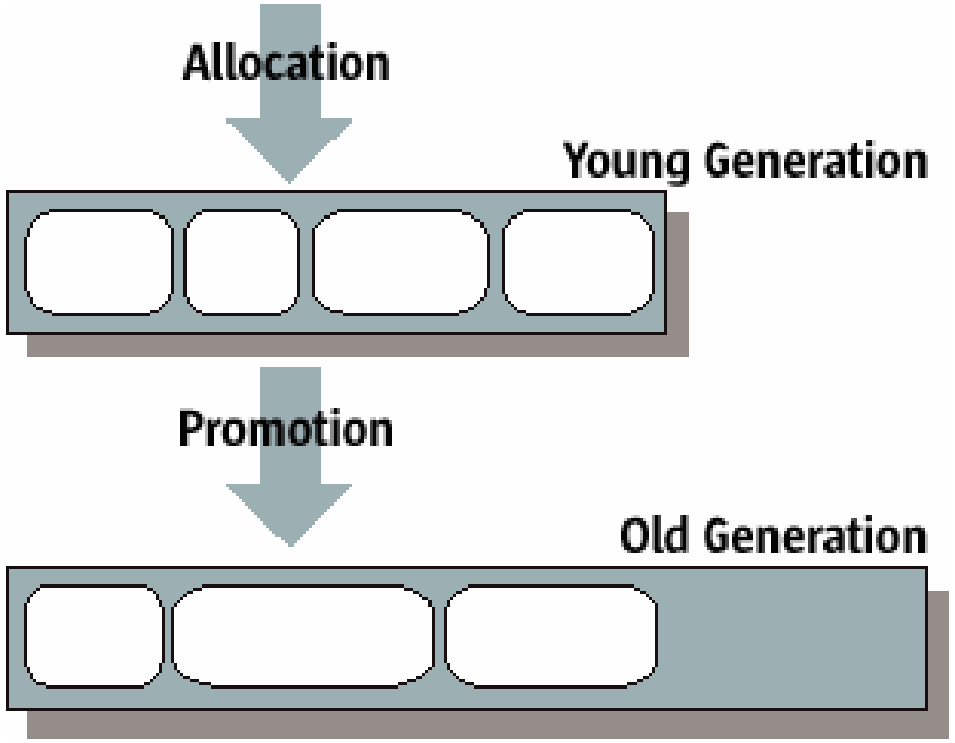
Characteristics of Young Generation Heap Space

- GC's occur relatively frequently
- GC's are efficient and fast because young generation space is usually small and likely to contain a lot of objects that are no longer referenced
- Objects that survive some number of young generation collections are promoted, or tenured, to old generation heap space

Characteristics of Old Generational Heap Space

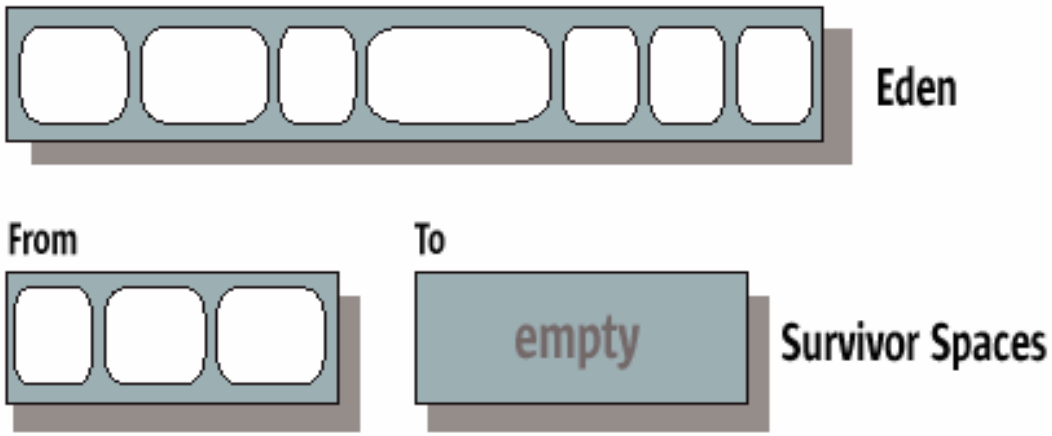
- Typically larger than young generation heap space
- Its occupancy grows more slowly
- GC's are infrequent but takes significantly longer time to complete

Generational GC



Young Generation Space Layout

- Made of an area called *Eden* plus two smaller survival spaces
- Most objects are initially allocated in Eden
- One of the two survival spaces hold objects that survived at least one young generation GC while the other is empty



GC Algorithms Used For

- Young generation
  - Algorithms used emphasize speed since young generation GC's are frequent
- Old generation
  - Algorithms used emphasize efficient space since old generation takes up most of the heap space and have to work with low garbage densities

When Does GC Occur?

When GC Occurs

- When the young generation fills up, a young generation GC occurs
  - Young generation GC is called minor GC
- When the old generation does not have enough space for the objects that are being promoted, a Full GC occurs
  - Full GC is also called major GC

GC Performance Metric

Important GC Performance Metric

- Throughput
  - The percentage of total time not spent in garbage collection, considered over long periods of time.
- Pause time
  - The length of time during which application execution is stopped while garbage collection is being performed

Application Requirement

- Different applications have different requirements
  - Higher throughput is more important for Web application: pauses during garbage collection may be tolerable, or simply obscured by network latencies.
  - Shorter pause time is more important to an interactive graphics application

Demo: Collecting GC Information through VisualGC

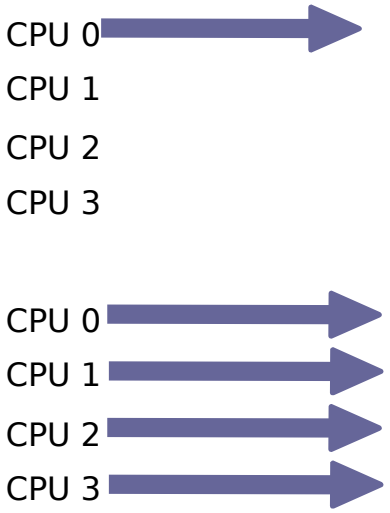
GC Algorithms

Choices of GC Algorithms (Three Different Comparison Criteria)

- Serial vs. Parallel
- Stop-the-world vs. Concurrent
- Compacting vs. Non-compacting vs. Copying

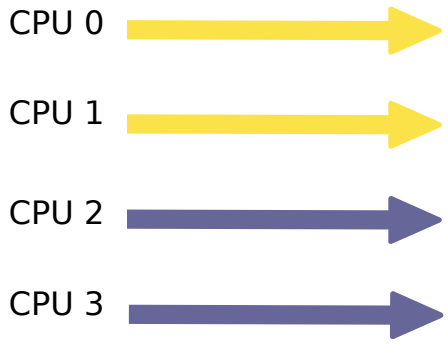
Serial vs. Parallel

- Serial
  - One CPU handles GC task
- Parallel
  - Multiple CPUs handle GC task simultaneously



Stop-the-world vs. Concurrent

- Stop-the-world
  - Execution of the application is completely suspended during GC
  - Simpler to implement (pro)
  - Longer pause time (con)
- Concurrent
  - One or more GC tasks can be executed concurrently with the application
  - Shorter pause time (pro)
  - Some overhead (con)



Compacting vs. Non-compacting vs. Copying

- Compacting
  - Move all live objects together and completely reclaim the remaining memory
- Non-compacting
  - Releases the space “in-place”
- Copying
  - Copies objects to a different memory area
  - The source area then becomes empty and available for fast and easy subsequent allocations

Types of GC Collector

Types of GC Collector

- Serial GC (Serial Collector)
- Parallel GC (Parallel Collector)
- Parallel Old GC (Parallel Compacting Collector)
- Concurrent Mark-Sweep (CMS) Collector

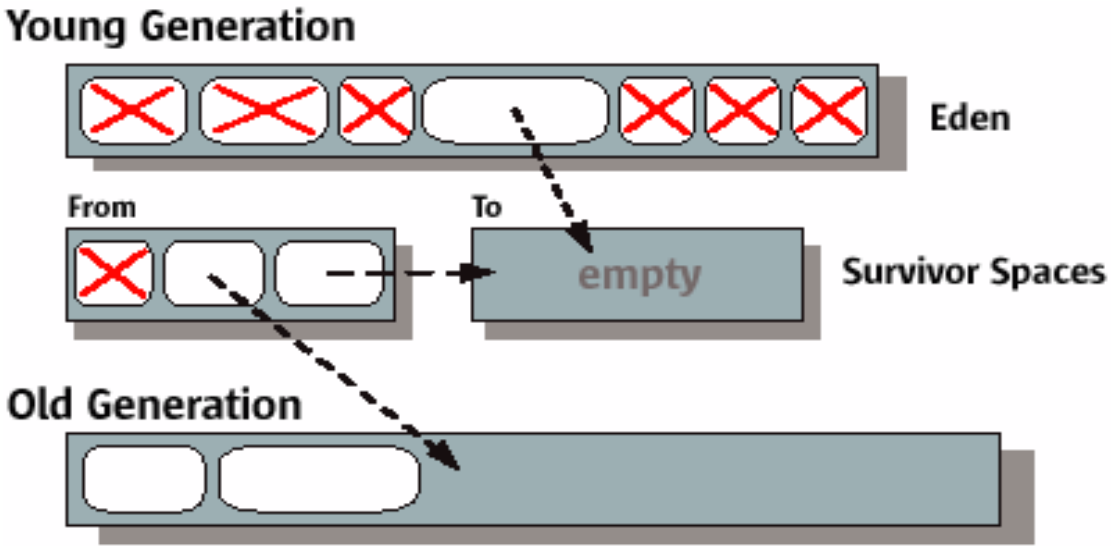
Serial GC

Serial GC

- Both young generation GC and old generation GC are done serially (using a single CPU)
- Stop-the-world
- Used for most applications that
  - are running on client-style machines
  - do not have low pause time requirement
- Default for client-style machines in Java SE 5 & 6
- Can be explicitly requested with
  - *-XX:+UseSerialGC*

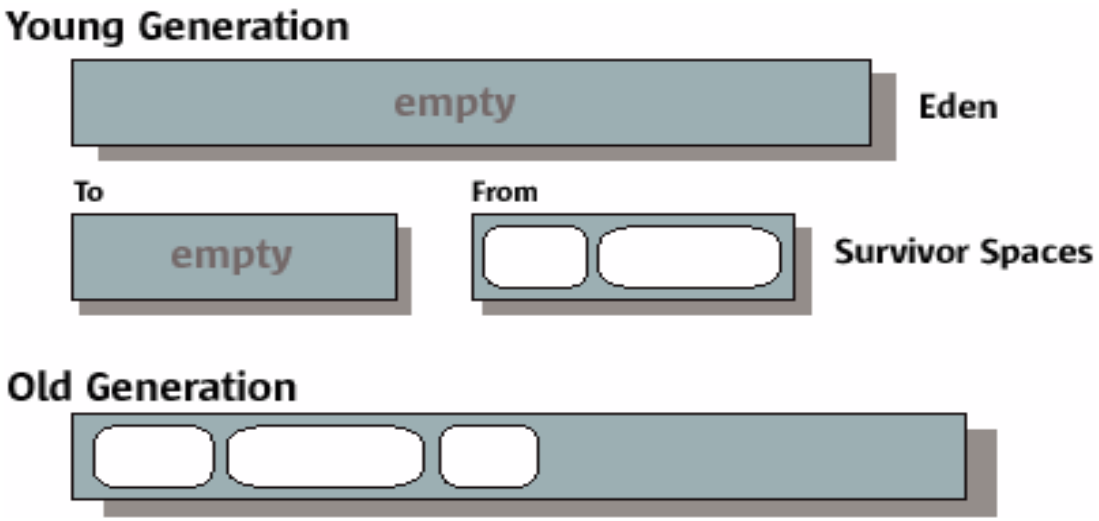
Serial GC on Young Generation: Before

- Live objects are copied from Eden to empty survival space
- Relatively young live objects in the occupied (From) survival space are copied to empty (To) survival space while relatively old ones are copied to old generation space directly



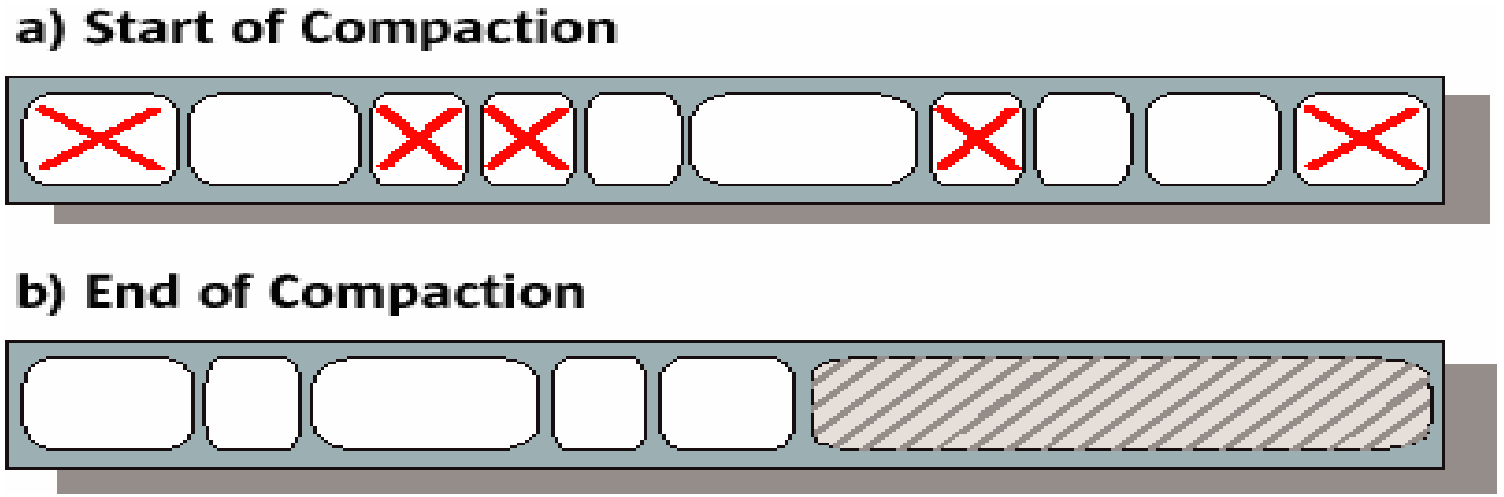
Serial GC on Young Generation: After

- Both Eden and the formerly occupied survival space are empty
- Only the formerly empty survival space contains live objects
- Survival spaces switch roles



Serial GC on Old Generation

- The old and permanent generations are collected via serial mark-sweep-compact collection algorithm



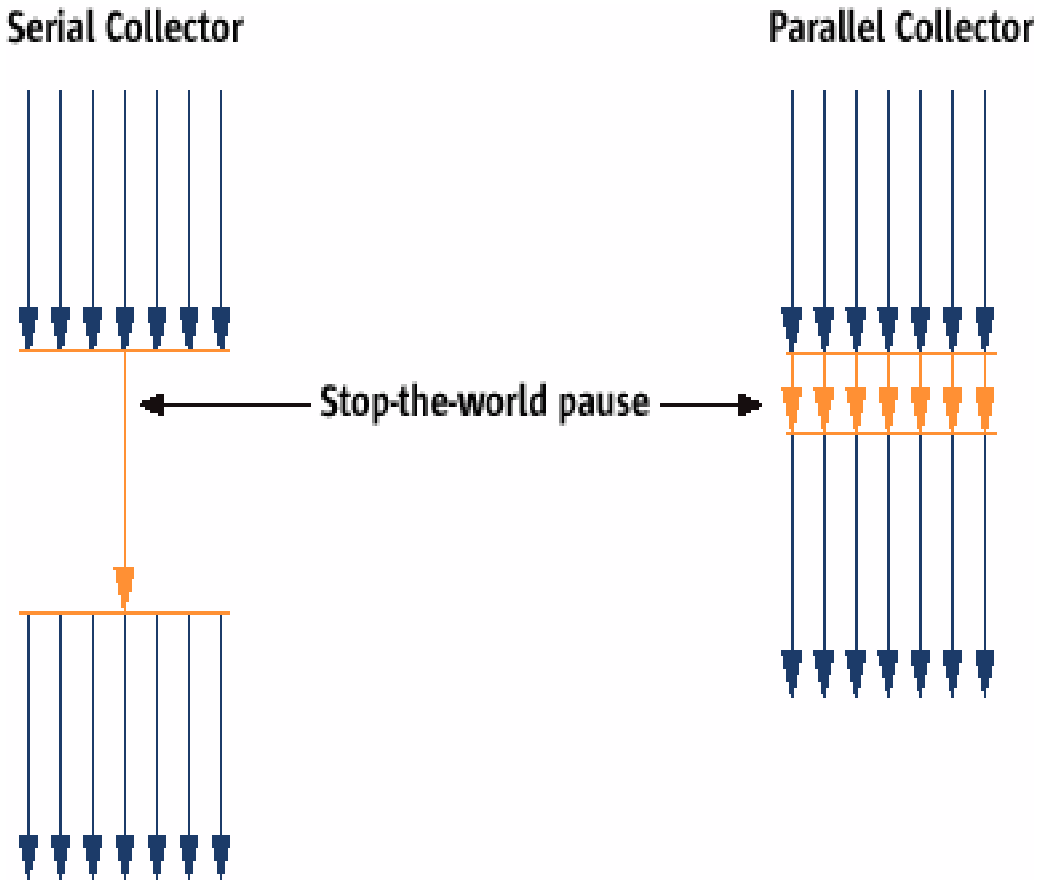
Parallel GC

Parallel GC

- Also known as Throughput Collector
  - Using multiple CPUs for young generation GC, thus increases the throughput
- Still stop-the-world – application is suspended
- Used for Java applications which run on machines with a lot of physical memory and multiple CPUs and do not have low pause time requirement
  - Infrequent but potentially long old generation GC can still occur
  - Examples: Batch applications, billing, payroll, scientific computing, etc
- Can be explicitly requested with
  - *-XX:+UseParallelGC*

Parallel GC on Young Generation

- Uses a parallel version of young generation collection algorithm of a serial collector



Parallel GC on Old Generation

- The old and permanent generations are collected via serial mark-sweep-compact collection algorithm (as in the Serial GC)

Parallel Old GC (Parallel Compact Collector)

Parallel Old GC

- It uses a new algorithm for old generation collection
  - Difference from the Parallel Collector
- Still stop-the-world
- Used for Java applications which run on machines with a lot of physical memory and multiple CPUs and do have low pause time requirement
  - Parallel operations of old generation collection reduces pause time
- Can be explicitly requested with
  - *-XX:+UseParallelOldGC*

Parallel Old GC on Young Generation

- It uses the same algorithm for young generation collection as Parallel GC

Parallel GC on Old Generation

- Three phases
  - Marking phase
  - Summary phase
  - Compaction phase
- Still stop the world

Concurrent Mark and Sweep (CMS) Garbage Collector: Regular Mode

CMS Collector

- Used when an application needs shorter pause time and can afford to share processor resources with GC when application is running
  - Applications with relatively large set of long-lived data (a large old generation) and run multi-CPU machines
  - Example: Web server
- Also called Low Pause Collector
- Can be explicitly requested with
  - *-XX:+UseConcMarkSweepGC*
  - *-XX:ParallelCMSThreads=<n>*

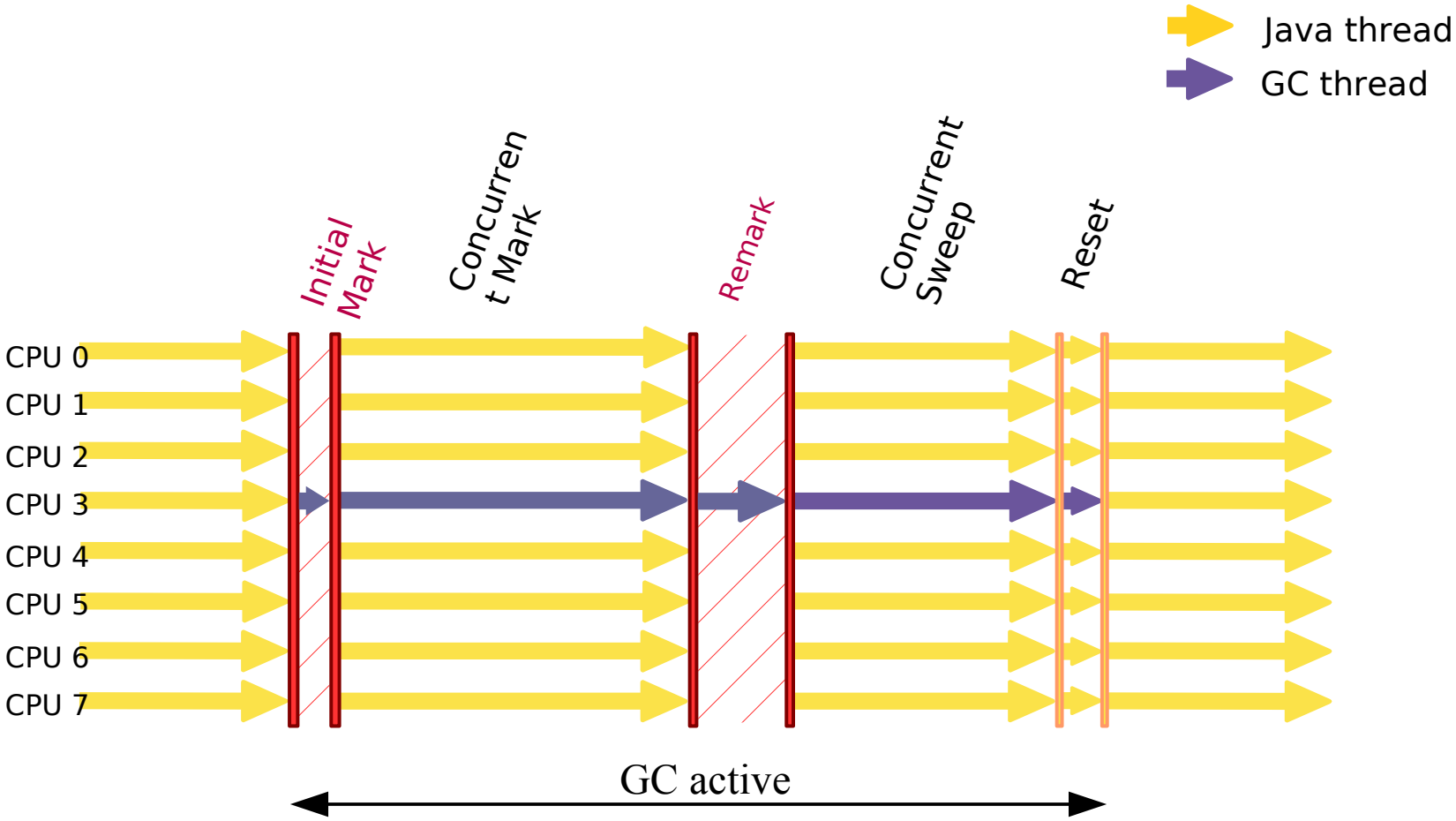
CMS Collector on Young Generation

- It uses the same algorithm for young generation collection as Parallel GC

CMS Phases (for Old Generation GC)

- Initial mark
  - Stop-the-world pause to mark from roots
  - Not a complete marking—only one level deep
- Concurrent mark
  - Mark from the set of objects found during Initial Mark
- Remark
  - Stop-the-world pause to complete marking cycle
  - Ensures a consistent view of the world
- Concurrent Sweep
  - Reclaim dead space, adding it back onto free lists
- Concurrent Reset

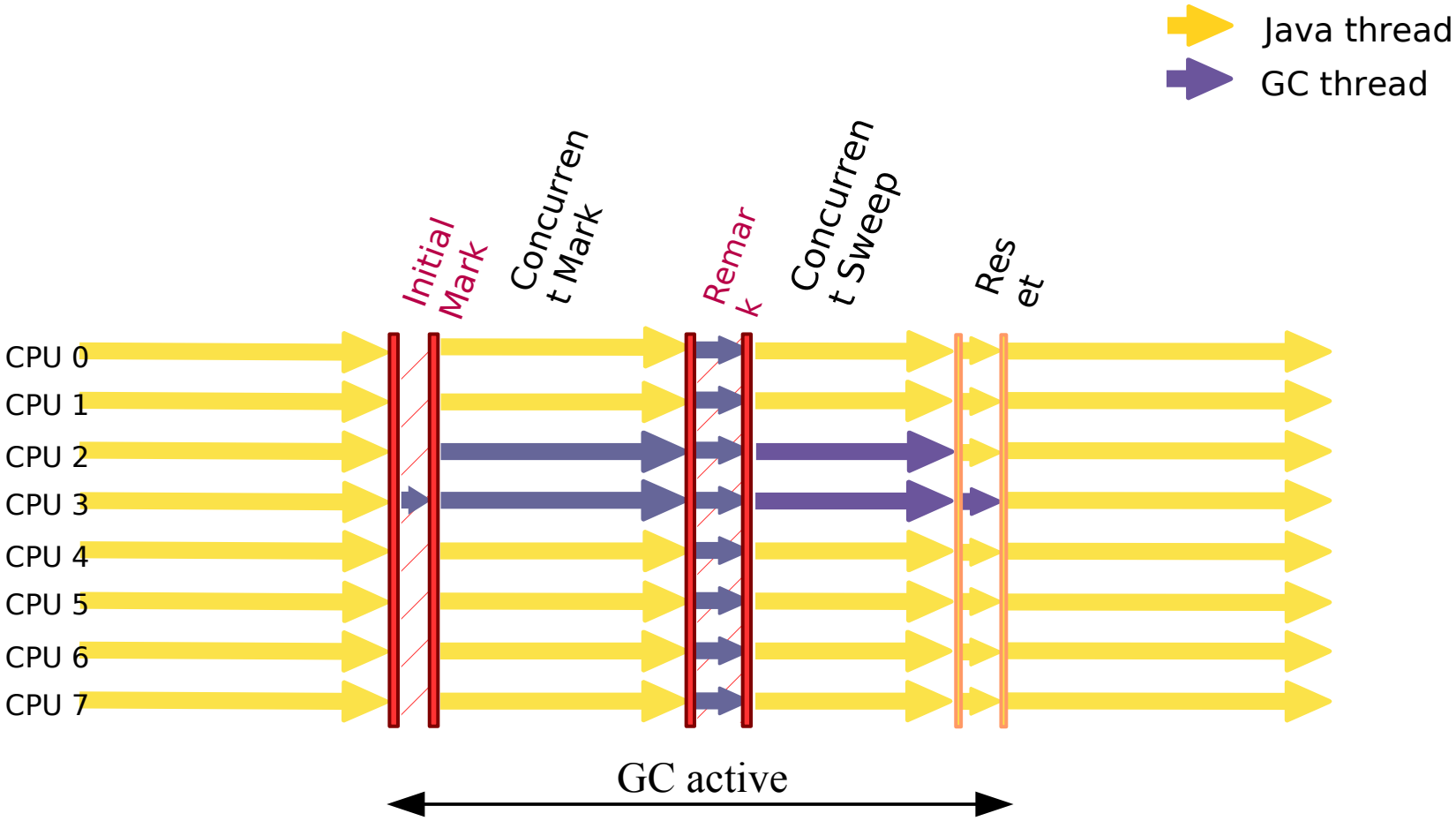
CMS Phases


Java thread
GC thread
Initial Mark
Concurrent Mark
Remark
Concurrent Sweep
Reset
CPU 0
CPU 1
CPU 2
CPU 3
CPU 4
CPU 5
CPU 6
CPU 7
GC active

CMS with Parallelism

- Remark is typically the largest pause
  - Often larger than Young GC pauses
  - Parallel remark available since Java 5 platform
- Single marking thread
  - Can keep up with ~4–8 cpus, usually not more
  - Parallel concurrent mark available in Java 6 platform
- Single sweeping thread
  - Less of a bottleneck than marking
  - Parallel concurrent sweep coming soon

CMS Phases with Parallelism



Concurrent Mark Sweep Collector

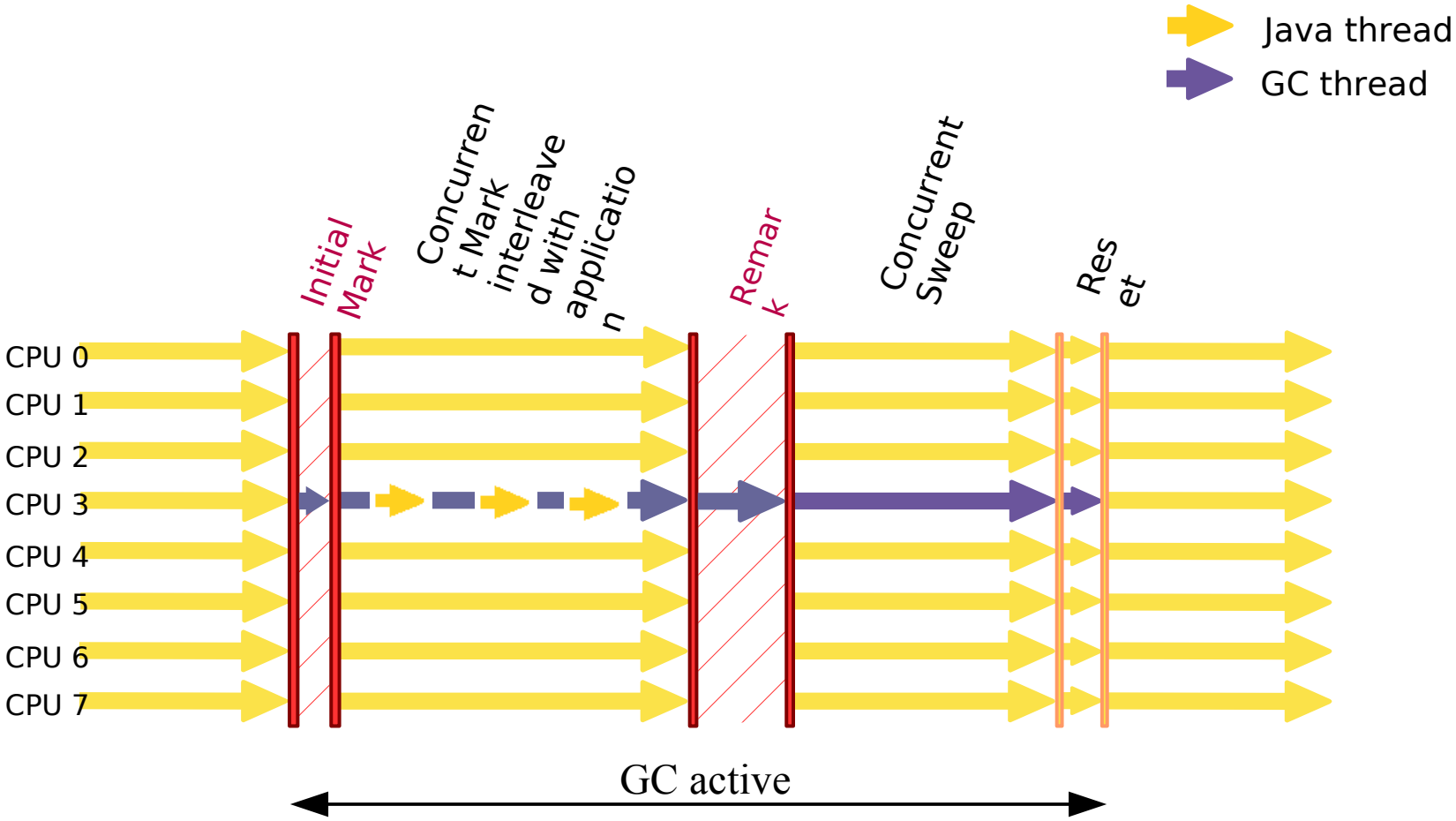
- Scheduling of collection handled by GC
  - Based on statistics in JVM
  - Or Occupancy level of tenured generation
    - -XX:CMSTriggerRatio
    - -XX:CMSInitiatingOccupancyFraction

Concurrent Mark and Sweep (CMS) Garbage Collector: Incremental Mode

CMS Collector in Incremental Mode

- CMS Collector can be used in a mode in which the concurrent phases are done incrementally
  - Meant to lesson the impact of long concurrent phases by periodically stopping the concurrent phase to yield back processing to the application
- Can be explicitly requested with
  - *-XX:+CMSIncrementalMode*

CMS Phases


Java thread
GC thread
Initial Mark
Concurrent Mark interleaved with application
Remark
Concurrent Sweep
Reset
CPU 0
CPU 1
CPU 2
CPU 3
CPU 4
CPU 5
CPU 6
CPU 7
GC active

Other Incremental CMS Options

- -XX:CMSIncrementalDutyCycle=<*n*%>
  - default: 50
- -XX:CMSIncrementalDutyCycleMin=<*n*%>
  - default: 10
- -XX:+CMSIncrementalPacing
  - default: true

Ergonomics

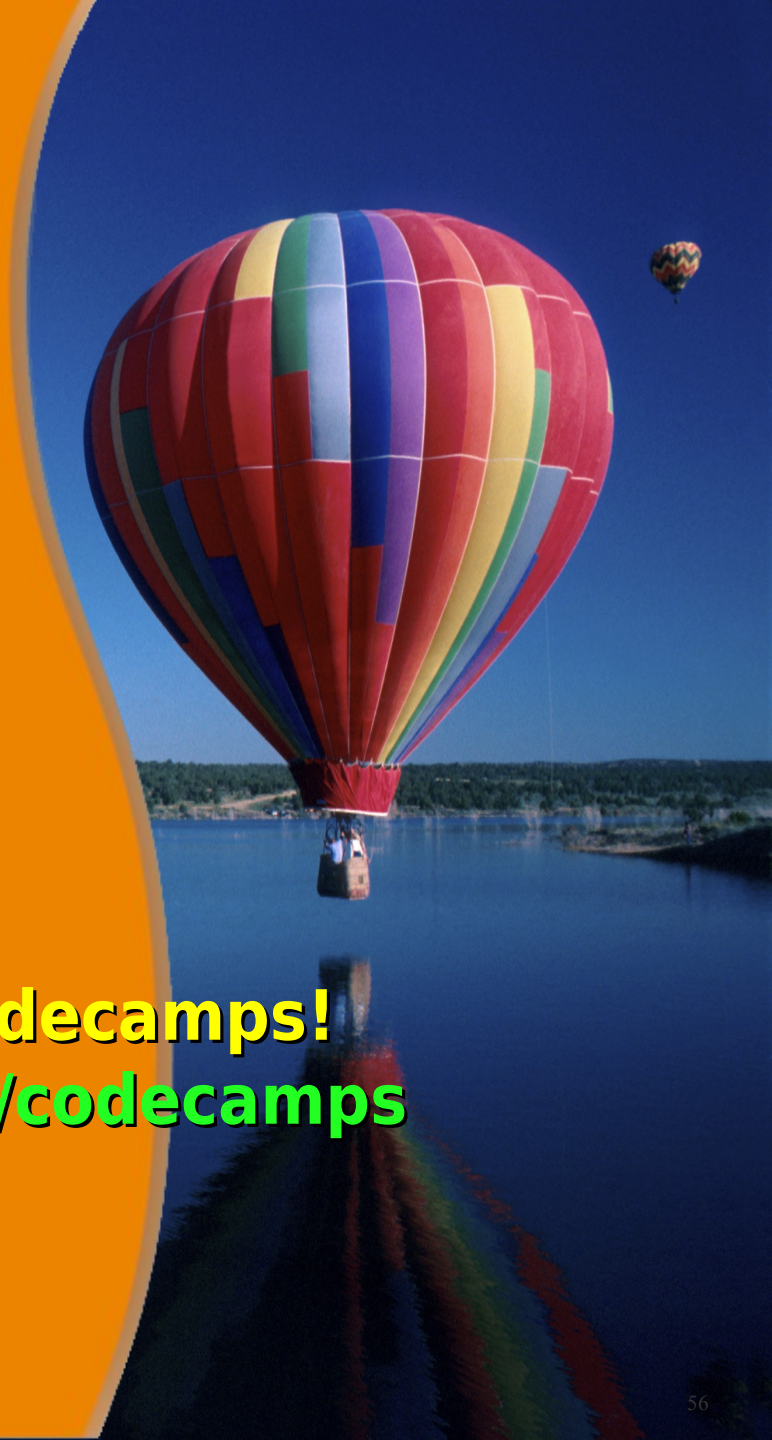
JVM Ergonomics
Throughput Collector

- Ergonomics enables the following:
  - Throughput garbage collector and Adaptive Sizing
  - (-XX :+UseParallelGC -XX:+UseAdaptiveSizePolicy)
  - Initial heap size of 1/64 of physical memory up to 1Gbyte
  - Maximum heap size of 1/4 of physical memory up to 1Gb
  - Server runtime compiler (-server)
- To enable server ergonomics on 32-bit Windows, use the following flags:
  - -server -Xmx1g -XX:+UseParallelGC
  - Varying the heap size.

Using JVM Ergonomics

- Maximum pause time goal
  - -XX:MaxGCPauseMillis=<*nnn*>
  - This is a hint, not a guarantee
  - GC will adjust parameters to try and meet goal
  - Can adversely effect application throughput
- Throughput goal
  - -XX:GCTimeRatio=<*nnn*>
  - GC Time : Application time = 1 / (1 + nnn)
  - e.g. -XX:GCTimeRatio=19 (5% of time in GC)
- Footprint goal
  - Only considered if first two goals are met

Thank you!



Check JavaPassion.com Codecamps!
http://www.javapassion.com/codecamps