



Performance Tuning: GC Friendly Java Programming

Simon Ritter

Technology Evangelist

Sun Microsystems, Inc.

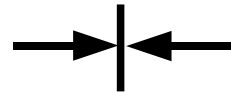


Generational Garbage Collection

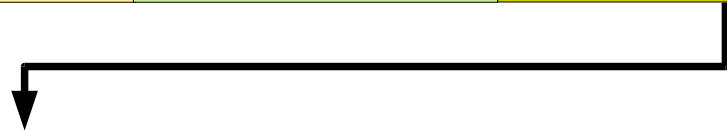
- Keeps young and old objects separately
 - > In spaces called **generations**
- The weak generational hypothesis
 - > Most new objects will die young
 - > Concentrate effort on young generation
 - > Need to keep track of old-to-young pointers
 - > Reference update tracking on old objects (write barrier)
 - > Eventually, have to also collect the old generation
- Different GC algorithms for each generation
 - > *“Use the right tool for the job”*

HotSpot VM Heap Layout

Survivor Ratio



Young Generation



Old Generation



Permanent Generation

Parallel Copy Collector

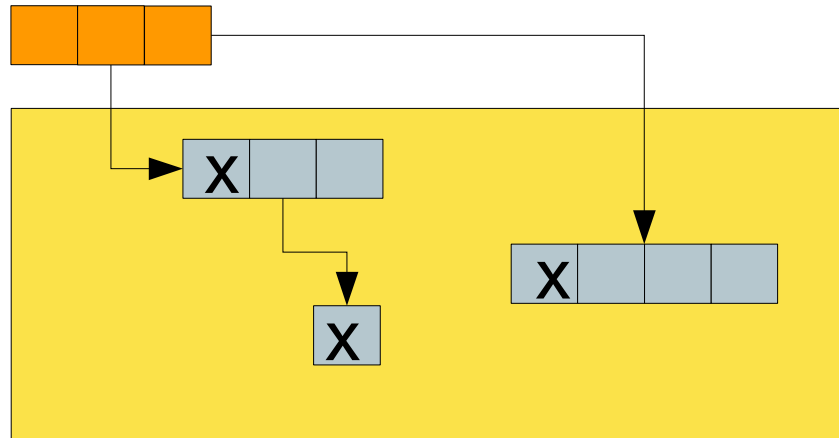
- Young generation
- Similar to copy-collector
 - > Still stop-the-world
- Allocates as many threads as CPUs
 - > Algorithm optimized to minimize contention
- Maximize work throughput
 - > Work stealing
- Potential locality of reference issue
 - > Each thread has separate destination in tenured space

Parallel Copy Collector

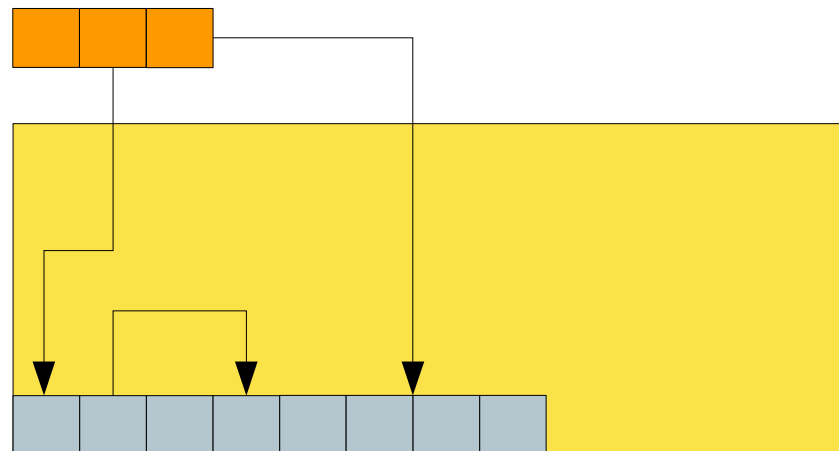
- `-XX:+UseParNewGC`
 - > Default copy collector will be used on single CPU machines
- `-XX:ParallelGCThreads=<num>`
 - > Default is number of CPUs
 - > Can be used to force the parallel copy collector to be used on single a CPU machine

Mark Sweep Compact GC

Old Generation



Before



After

Mark Sweep Compact GC

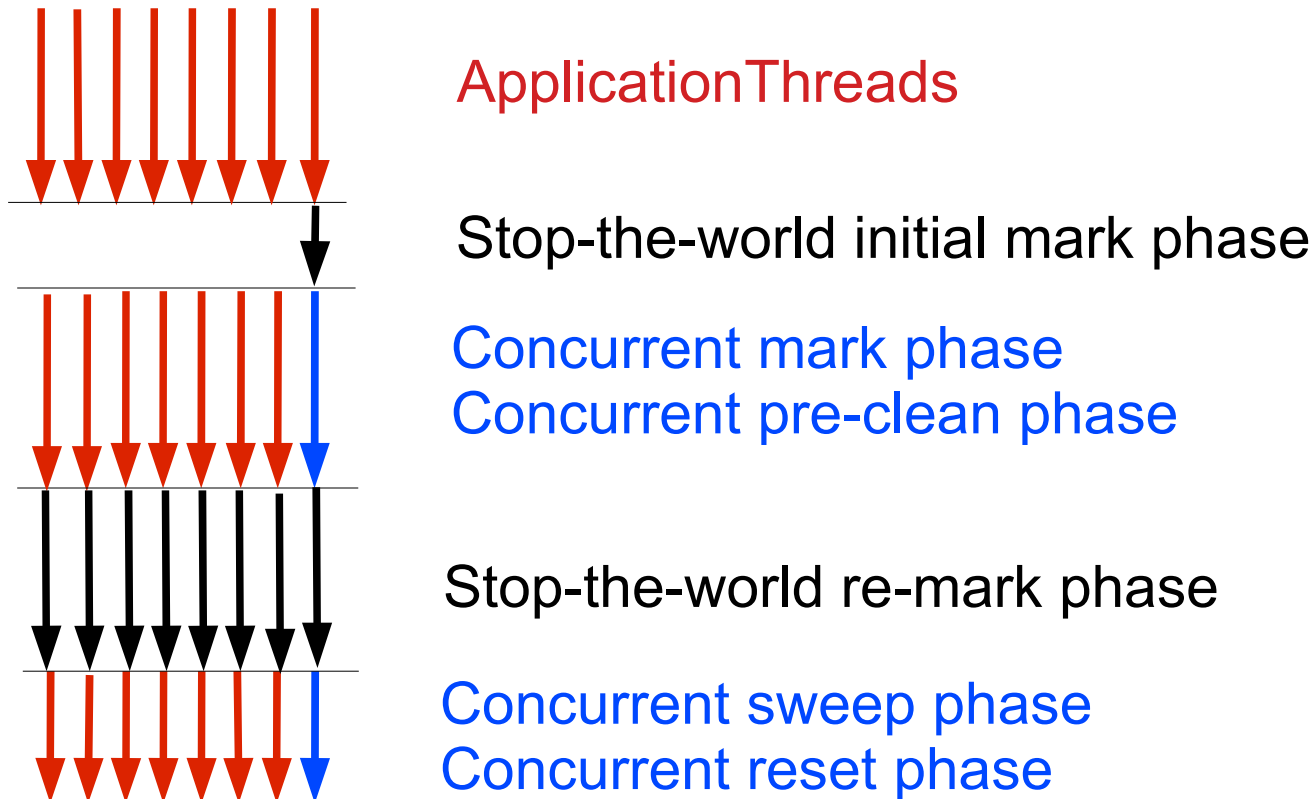
- Eliminates fragmentation issue of Mark-Sweep
- Allocation becomes stack-based
- Order of objects maintained
 - > Locality of reference
- Requires multiple passes to complete
 - > Mark live objects
 - > Compute new location
 - > Update pointers

Low Pause Collector

Concurrent Mark Sweep

-XX:+UseConcMarkSweepGC

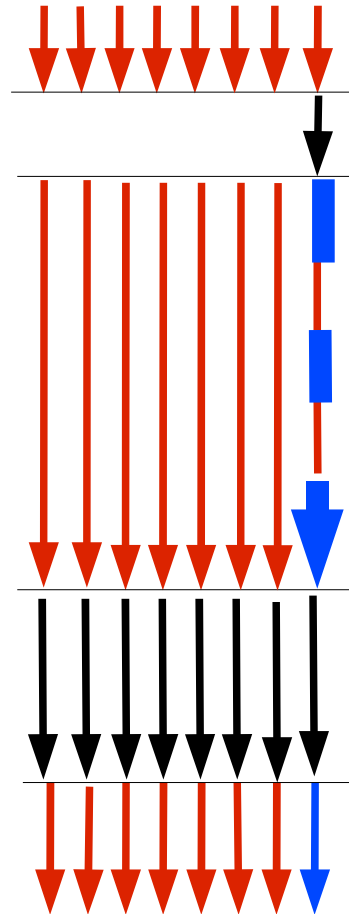
-XX:ParallelCMSThreads=<n>



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

Incremental CMS



Marking work
interleaved with
application work

Incremental CMS

- -XX:+CMSIncrementalMode (default: false)
- -XX:CMSIncrementalDutyCycle=<n%> (default: 50)
- -XX:CMSIncrementalDutyCycleMin=<n%> (default: 10)
- -XX:+CMSIncrementalPacing (default: true)
- DutyCycle of 10 and DutyCycleMin of 0 can help certain applications

Throughput Collector

Parallel Scavenge

- Stop-the-world
- Similar to parallel-copy collector
- Aimed at large young spaces
 - > 10Gb or larger
- Scales well with more CPUs
- Adaptive tuning policy
 - > Survivor ratio
- Promotion undo to prevent out of memory

Throughput Collector

Parallel Scavenge

- Available from JDK 1.4.1
- `-XX:+UseParallelGC`
- `-XX:ParallelGCThreads=<num>`
 - > Control number of threads
- `-XX:+UseAdaptiveSizePolicy`
 - > Automatically sizes the young generation and selects optimum survivor ratio

Performance Improvements 5.0_06

- Disable inline caches
 - > Avoid cache pollution of L1 core cache
- Use jump tables (-XX:+UseJumpTables)
 - > SPARC only
 - > Instead of a series of conditional branches
- Align method entry points and loop tops on single instruction boundary
- UltraSPARC T1 detection by HotSpot
 - > Enables above optimizations

Performance Improvements 5.0_06

- Biased Locking
 - > -XX:+UseBiasedLocking
 - > -XX:+TraceBiasedLocking (for debugging)
 - > Good for applications with lots of uncontended synchronization
 - > Can have negative impact for certain locking patterns

Performance Improvements 5.0_06

- Parallel Old Generation GC
 - > -XX:+UseParallelOldGC
- Uses mark-sweep-compact algorithm
- Not all phases are currently parallel
- Can be good for UltraSPARC T1
 - > CMS is single threaded
 - > CMS cannot keep up with mutator threads
 - > Use parallel old, assuming pause times are acceptable

Large Page Sizes

- `-XX:+UseLargePages`
 - > Cross platform (only on by default for Solaris)
 - > Improves utilisation of TLB
 - > Kernel support in Linux 2.6 and Windows 2003 server
- `-XX:LargePageSizeInBytes=<n>`
 - > 8m SPARC
 - > 4m x86
 - > 2m x86_64
 - > 256m supported for UltraSPARC T1

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 applicaiton 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

Heap Tuning Beyond Ergonomics

- Increase heap size
 - > Ergonomics chooses up to 1GB, some applications need more memory for high performance
 - > -Xms3g -Xmx3g
- Increase the size of the young generation
 - > Generally: $\frac{1}{4}$ to $\frac{1}{2}$ the overall heap size
 - > Sizing above $\frac{1}{2}$ the overall heap size is supported
 - > Only makes sense with throughput collector

Tuning For CMT Systems

- Limit Parallel GC thread count
 - > `-XX:ParallelGCThreads=20`
- Limiting GC thread count is important when running multiple JVMs on a single OS instance
 - > Total GC threads should not exceed HW thread count
- Experiment with Biased Locking
 - > Could improve performance

Tuning For CMT Systems

- Try using both young and old generation parallel collectors
- Combine parallel new generation collector with CMS
- `-XX:+MaxTenuringThreshold=31`
 - > When using CMS
 - > Unless `-XX:+UseBiasedLocking` is being used
- Try `-XX:+AggresssiveOpts`
- `-XX:CICompilerCount=<n>`
- `-XX:CompileThreshold`

Object Allocation (1/2)

- Typically, object allocation is very cheap!
 - > 10 native instructions in the fast common case
 - > No remembered set overhead on new objects
 - > C/C++ has faster allocation? Not!
- Reclamation of new objects is very cheap too!
 - > Young GCs in generational systems
- So
 - > Do not be afraid to allocate small objects for intermediate results
 - > GCs **love** small, immutable objects
 - > Generational GCs **love** small, short-lived objects

Object Allocation (2/2)

- **not** advising
 - > Needless allocation
 - > More frequent allocations will cause more frequent GCs
- **do** advise
 - > Using short-lived immutable objects instead of long-lived mutable objects
 - > Using clearer, simpler code with more allocations instead of more obscure code with fewer allocations

Large Objects

- Very large objects are:
 - > Expensive to allocate (maybe not through the fast path)
 - > Expensive to initialize (zeroing)
 - > Can cause performance issues
- Large objects of different sizes can cause fragmentation
 - > For non-compacting or partially-compacting GCs
- Avoid if you can
 - > And, yes, this is not always possible or desirable

Explicit GCs (1/2)

- Avoid them!
 - > Applications do not have enough information
 - > GC does (knows allocation/promotion rate, etc.)
 - > System.gc() at the wrong time
 - > Hurts performance with no benefit
- Exceptions
 - > Between well-defined application phases (maybe)
 - > When performance does not matter (e.g., late at night)
- Java HotSpot™ virtual machine
 - > System.gc() does a stop-the-world full GC
 - > Use -XX:+DisableExplicitGC to ignore System.gc()

Explicit GCs (2/2)

- Incremental GCs
 - > Designed to avoid full GCs...
 - > But `System.gc()` does exactly that!
- In the Java HotSpot virtual machine (CMS)
 - > `-XX:+ExplicitGCInvokesConcurrent`
- Beware
 - > Libraries that call `System.gc()`
 - > Run FindBugs over your libraries to check for that
 - > Java™ RMI calls `System.gc()` for its distributed GC algorithm
 - > Decrease its frequency, or invoke concurrent, or both!

Data Structure Sizing (1/2)

- Array-based data structures
 - > Avoid frequent re-sizing
- e.g., this will allocate the associated array twice

```
ArrayList<String> list = new ArrayList<String>();  
list.ensureCapacity(1024);
```

- The preferred version
 - > (Part of periodic audits of the Java Platform, Standard Edition (Java SE) libraries)

```
ArrayList<String> list = new ArrayList<String>(1024);
```

Data Structure Sizing (2/2)

- Additionally, try to size data structures as realistically as possible

```
ArrayList<String> list = new ArrayList<String>(1024);
```

- If 1M strings are added to it:
 - > Several array-resizing operations will take place
 - > They will allocate several large-ish arrays
 - > They will cause a lot of array copying
 - > They might cause fragmentation issues on non-compacting GCs

Object Pooling

- Legacy of older VMs with terrible allocation performance
- Remember
 - > Generational GCs **love** short-lived, immutable objects...
 - > Not long-lived, highly mutable objects
- Unused objects in pools
 - > Are like a bad tax
 - > Are live; the GC must process them
 - > Provide no benefit; the application does not use them

Object Pooling

- List of issues
 - > Sizing
 - > Too small: allocate anyway
 - > Too large: too much footprint overhead + pressure on GC
 - > Safety
 - > Reintroduce malloc/free mistakes
 - > Scalability
 - > Must allocate/de-allocate efficiently
 - > **synchronized** defeats the VM's fast allocation mechanism
 - > Compatibility
 - > Incompatible with most standard libraries

Object Pooling

- Exceptions
 - > Objects that are expensive to allocate and/or initialize
 - > Objects that represent scarce resources
 - > Examples
 - > Threads pools
 - > Database connection pools
- Caveats to the exceptions
 - > Use existing libraries wherever possible
 - > Can you write a better thread pool than Doug Lea?

Memory Leak Types

- “Traditional” memory leaks
 - > Heap keeps growing, and growing, and growing...
 - > OutOfMemoryError
- “Temporary” memory leaks
 - > Heap usage is temporarily very high, then it decreases
 - > Bursts of frequent Gcs
- Both finalizers and reference objects
 - > Can delay the reclamation of objects...
 - > As well as everything reachable from them
 - > Temporary heap usage spikes

Memory Leak Detection Tools

- Many tools to choose from
- *“Is there a memory leak?”*
 - > Monitor VM’s heap usage with jconsole and jstat
- *“Which objects are filling up the heap?”*
 - > Get a class histogram with jmap or
 - > -XX:+PrintClassHistogram and Ctrl-Break
- *“Why are these objects still reachable?”*
 - > Get reachability analysis with jhat

For More Information

- Memory management white paper
 - > <http://java.sun.com/j2se/reference/whitepapers/>
- Destructors, Finalizers, and Synchronization
 - > <http://portal.acm.org/citation.cfm?id=604153>
- Finalization, Threads, and the Java Technology Memory Model
 - > <http://developers.sun.com/learning/javaoneonline/2005/coreplatform/TS-3281.html>
- Memory-retention due to finalization article
 - > <http://www.devx.com/Java/Article/30192>



Performance Tuning: GC Friendly Java Programming

Simon Ritter

Technology Evangelist

Sun Microsystems, Inc.

