Concurrent and Parallel Garbage Collection

Michael Hicks

CMSC 433, Fall 2013

University of Maryland College Park

Thanks to Richard Jones for many of these slides

Computing Laboratory, University of Kent, Canterbury

Recall Amdahl's Law

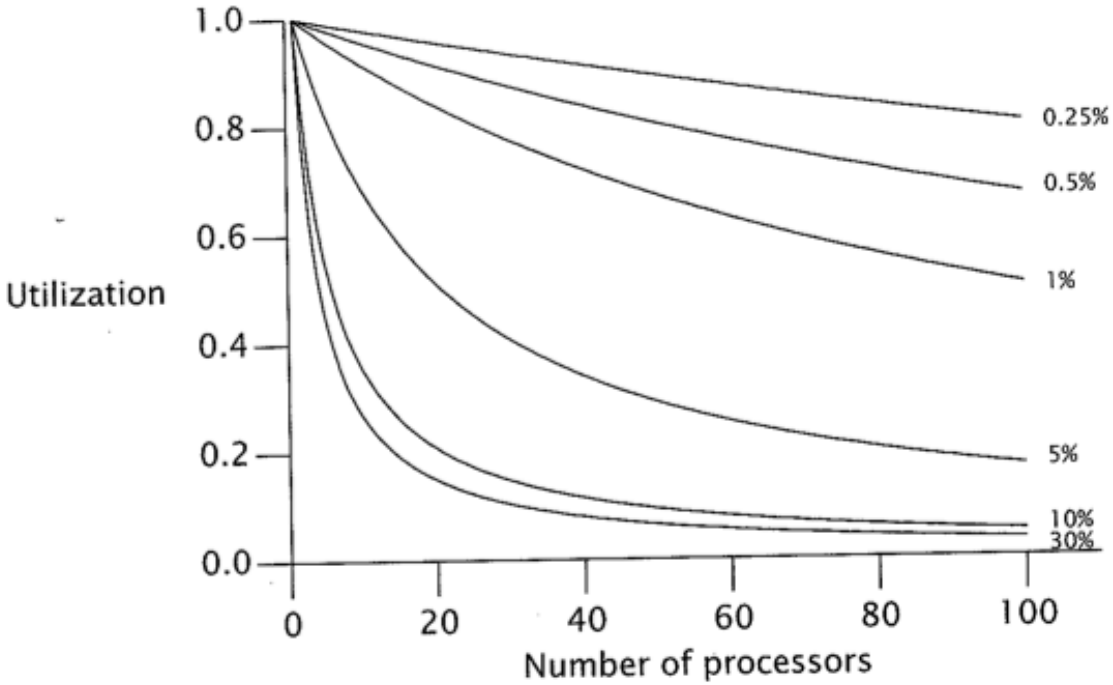
Speedup ≤ 1 / (F + (1-F) / N)

- where Speedup = ST time / MT time

Utilization = Speedup / N

N is the # processors

F is fraction of work that is serial



Takeaway message:

Want to reduce the amount of serial work performed!

Memory Management

1. Each call to `new Foo(…)` allocates to the heap
2. Eventually we run out of space, and must run the garbage collector

Thus, two potential contributors to F

- **Allocation:** The data structures used to manage memory need to be thread safe, and thus allocation could incur synchronization cost
  - Programs with a lot of allocation will lose parallelism
- **GC:** The garbage collector may "stop the world" and thus it contributes to F (rather than being just an overhead in each thread)

One idea: make each individually faster

Algorithmic terms

Tracing

The process of starting at the roots of your program and following pointers to find reachable memory in the heap (e.g., using BFS or DFS)

Roots

The elements of the heap immediately addressable/live from the current collection point, e.g., global variables, local variables (registers)

Marking and Sweeping

Action performed with tracing to mark live objects; action performed to traverse entire heap, to throw away non-marked objects

Graphical notation

- Black node: fully processed
- Gray node: queued for consideration
- White node: not processed at all
  - Following GC, all white nodes are garbage

Basic algorithms

—The garage metaphor—

Reference counting: Maintain a note on each object in your garage, indicating the current number of references to the object. When an object's reference count goes to zero, throw the object out (it's dead).

Mark-Sweep: Put a note on objects you need (roots). Then recursively put a note on anything needed by a live object.
Afterwards, check all objects and throw out objects without notes.

Mark-Compact: Put notes on objects you need (as above). Move anything with a note on it to the back of the garage.
Burn everything at the front of the garage (it's all dead).

Copying: Move objects you need to a new garage. Then recursively move anything needed by an object in the new garage.
Afterwards, burn down the old garage (any objects in it are dead)!

Generational GC

Weak generational hypothesis

"Most objects die young" [Ungar, 1984]

Common for 80-95% objects to die before a further MB of allocation.

Strategy:

- Segregate objects *by age* into **generations** (regions of the heap).
- Collect different generations at different frequencies.
  - so need to "remember" pointers that cross generations.
- Concentrate on the nursery generation to reduce pause times.
  - full heap collection pauses 5-50x longer than nursery collections.

Generational GC in practice

Typical configuration involves two generations

- A nursery region providing fast, bump-pointer allocation.
- A mature region managed by a mark-sweep collector.
  - requires occasional compaction

HotSpot VM has three generations

- Nursery region is really three spaces
  - “eden”, “from” (aka survivor 0), and “to” (aka survivor 1) – collection from the former two into the latter. When the latter is full, collect into old region
- Mature region is called “old” or “tenured”
- One additional region called “perm” which is never collected
  - Requires special handling

How does this help?

Study by Blackburn, Cheng, and McKinley (2004) found

- Bump-pointer allocation is indeed fast (reduces allocation time) and also improves cache locality (credits allocation time)
- Generational GC indeed reduces overall time in GC
  - Together, these both improve F

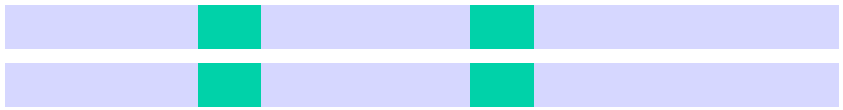
How can we reduce F even further? Exploit parallelism!

- Concurrent allocation
- Concurrent marking
- Parallel marker and collector threads
- Concurrent sweeping
- Parallel & concurrent compaction

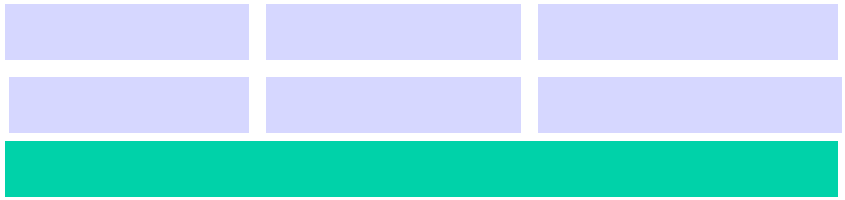
Terminology

Single threaded collection

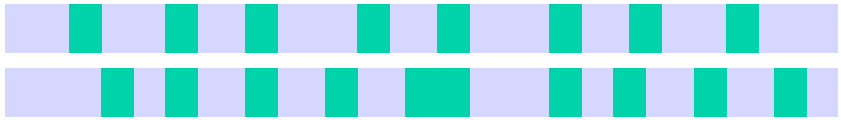
Parallel collection

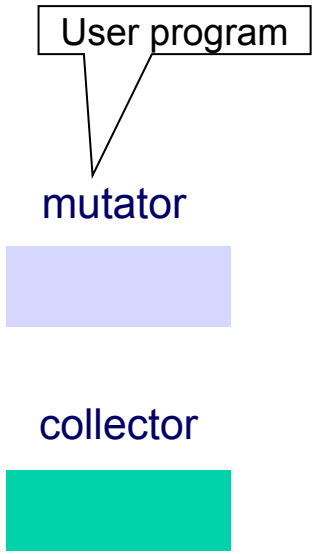


Concurrent collection



Incremental collection

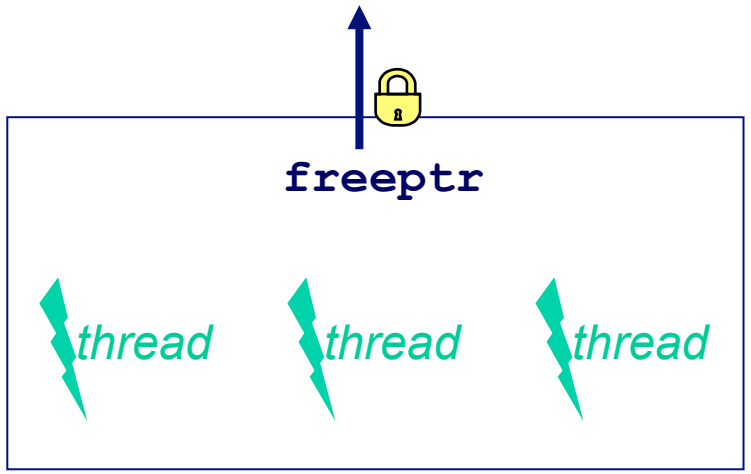




A. Concurrent allocation

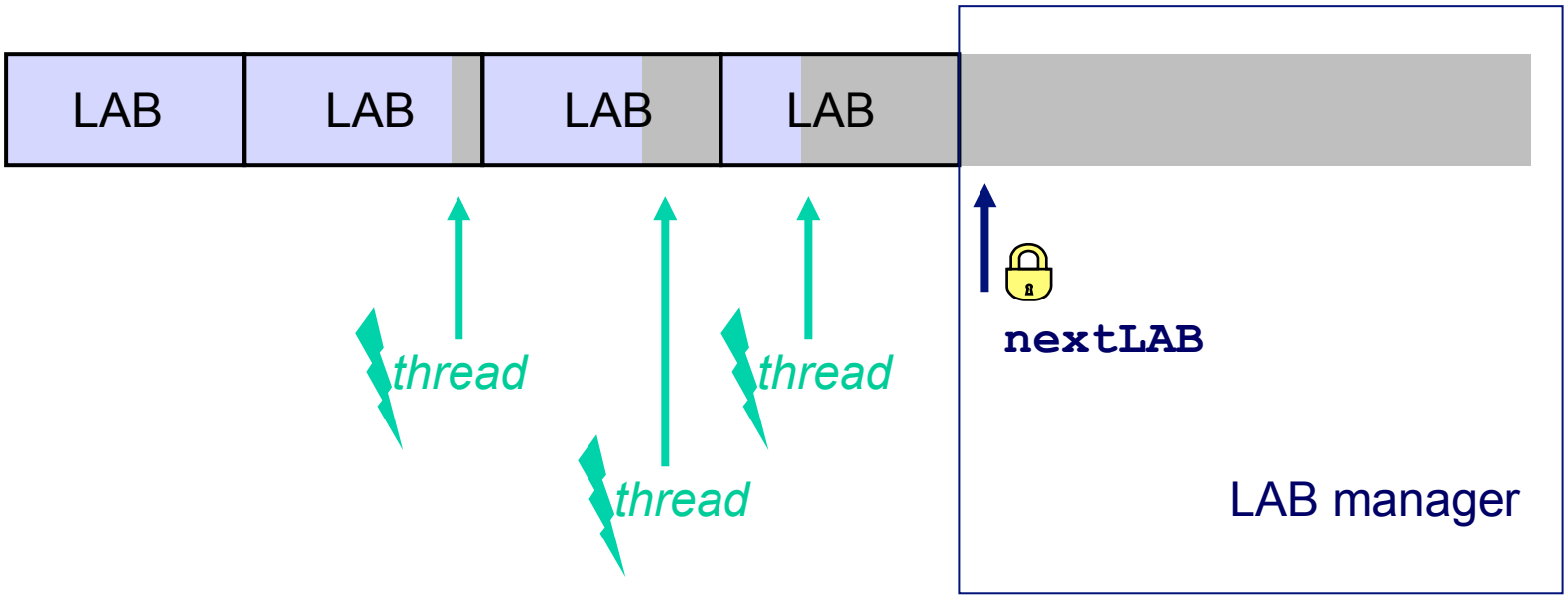
Multiple user threads

- Must avoid contention for the heap
- Avoid locks, avoid atomic instructions (CAS)



Local Allocation Blocks

- Thread contends for a contiguous local allocation block (LAB) — CAS.
- Thread allocates within LAB (bump a pointer) — no contention.
- Locality properties make this effective even for GCs that rarely move objects — needs variable-sized LABs.



Activate in HotSpot using -XX:+UseTLAB

Concurrent GC: issues

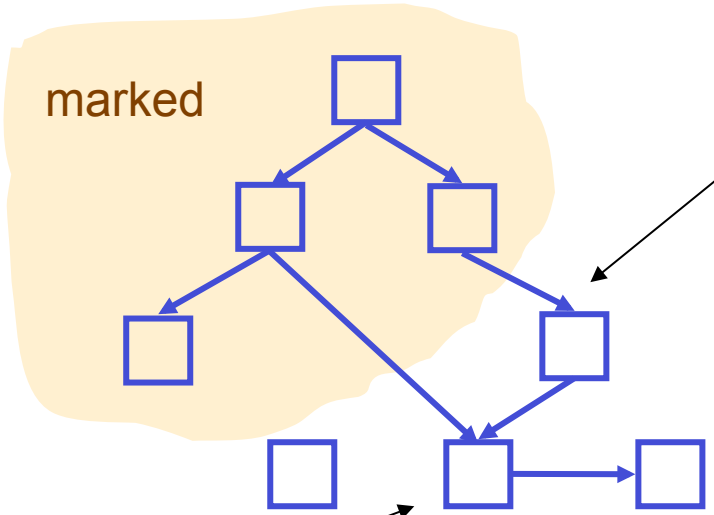
What can go wrong if the GC runs in parallel with the mutator?

Two problems

- Problem 1: Mutator modifies the contents of an object already considered (i.e., a black node, pointing to white node)
  - To happen, a pointer to a white object must be written to a black object
  - The original pointer to the white object must be lost
- Problem 2 (copying/compacting collectors only): GC moves an object the mutator wants to access (a black or gray node)

B: Concurrent Marking

Tracing concurrently with user threads introduces a coherency problem: the mutator might hide pointers from the collector.



Snapshot at the beginning write barrier

Record & revisit

Incremental update write barrier

Record & revisit

IU catch changes to connectivity.
SAB prevent loss of the original path.

Write barrier properties

Performance: no barriers on local variables

Incremental update

- Mutator can hide live objects from the collector
  - Non-atomic ops in order: update, then “shade” (not the reverse)
- Need a final, stop the world, marking phase.

Snapshot at the beginning

- Any object reachable at the start of the phase remains reachable.
- So no STW phase required.
- But more floating garbage retained than for incremental update.

What about allocation?

One approach: allocate all objects as black or grey

- Conservative, since objects may die before end of cycle
- But objects need not be traversed

Alternative: allocate objects as white

- May die before end of cycle
- But must be sure we traverse them (how?)

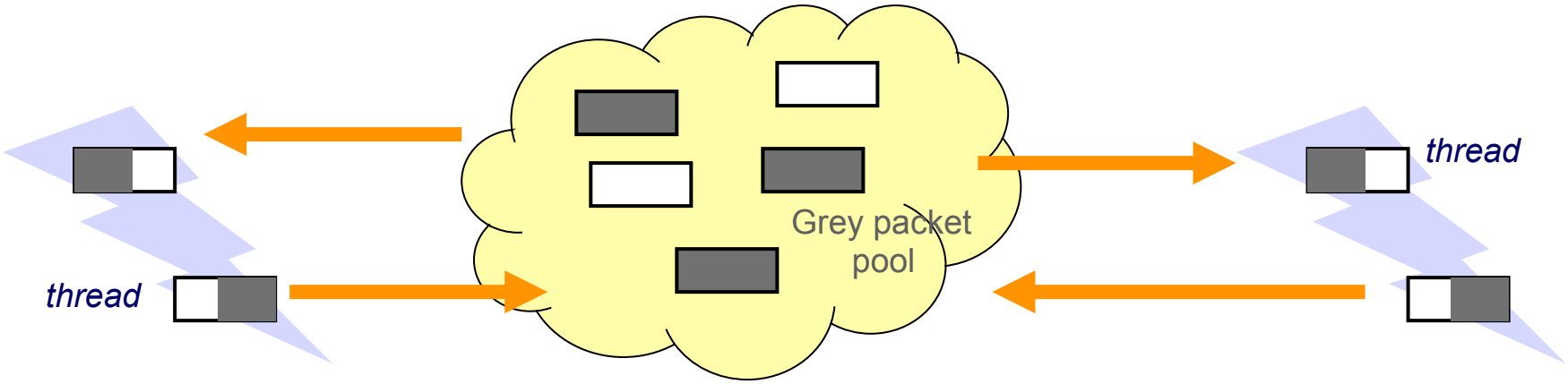
C. Parallel markers

The goal is always to avoid contention (e.g. for the mark stack).

Thread-local mark stacks, work-stealing.

Grey packets.

Grey packets



Marker threads acquire a full packet of marking work (grey references).

They mark & empty this packet and fill a fresh (empty) packet with new work.

The new, filled packet is returned to the pool.

- Avoids most contention
- Simple termination.
- Allows prefetching (unlike traditional mark stack).
- Reduces processor weak ordering problems (fences only around packet acquisition/disposal).

E. Compaction

Without compaction

- Heaps tend to fragment over time.

Issues for compaction:

- Moving objects.
- Updating all references to a moved object.
- In parallel
- And concurrently.

Read barrier methods

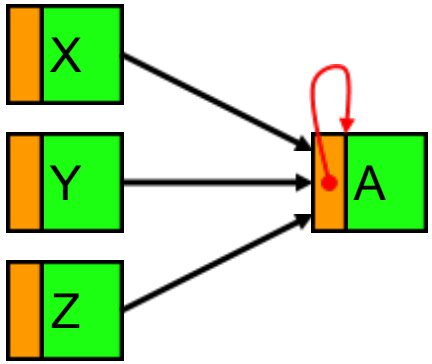
Idea: don't let the mutator see objects that the collector hasn't seen.

How?

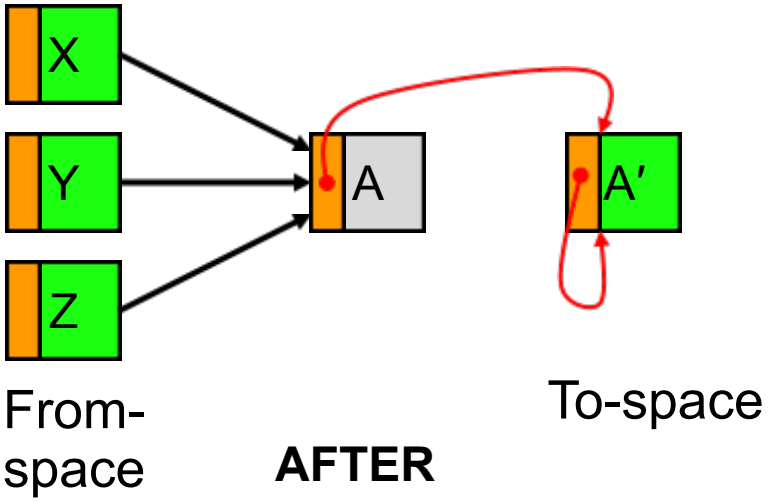
- Trap mutator accesses and mark or copy & redirect.
- Read barriers
  - software
  - memory protection.

Read barrier: To-space invariant

- Problem: Collector moves objects (defragmentation)
  - Mutator is finely interleaved
- Solution: read barrier ensures consistency
  - Each object contains a forwarding pointer [Brooks]
  - Read barrier unconditionally forwards all pointers
  - Mutator never sees old versions of objects
- Will the mutator utilization have any effects because of the read barrier ?



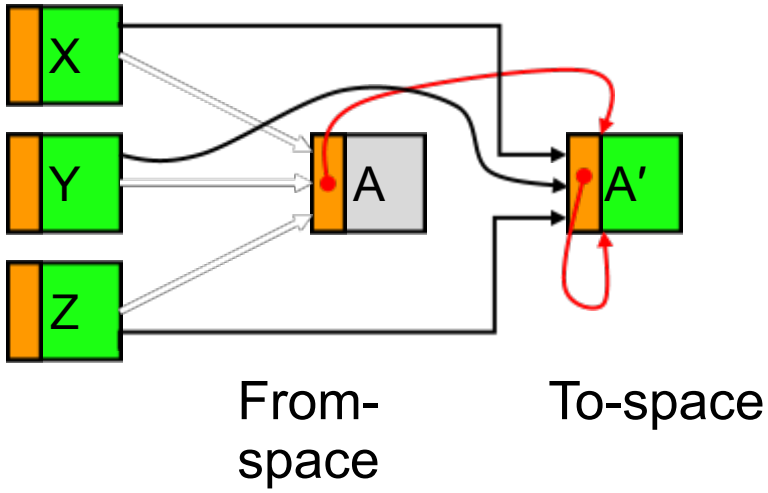
BEFORE



AFTER

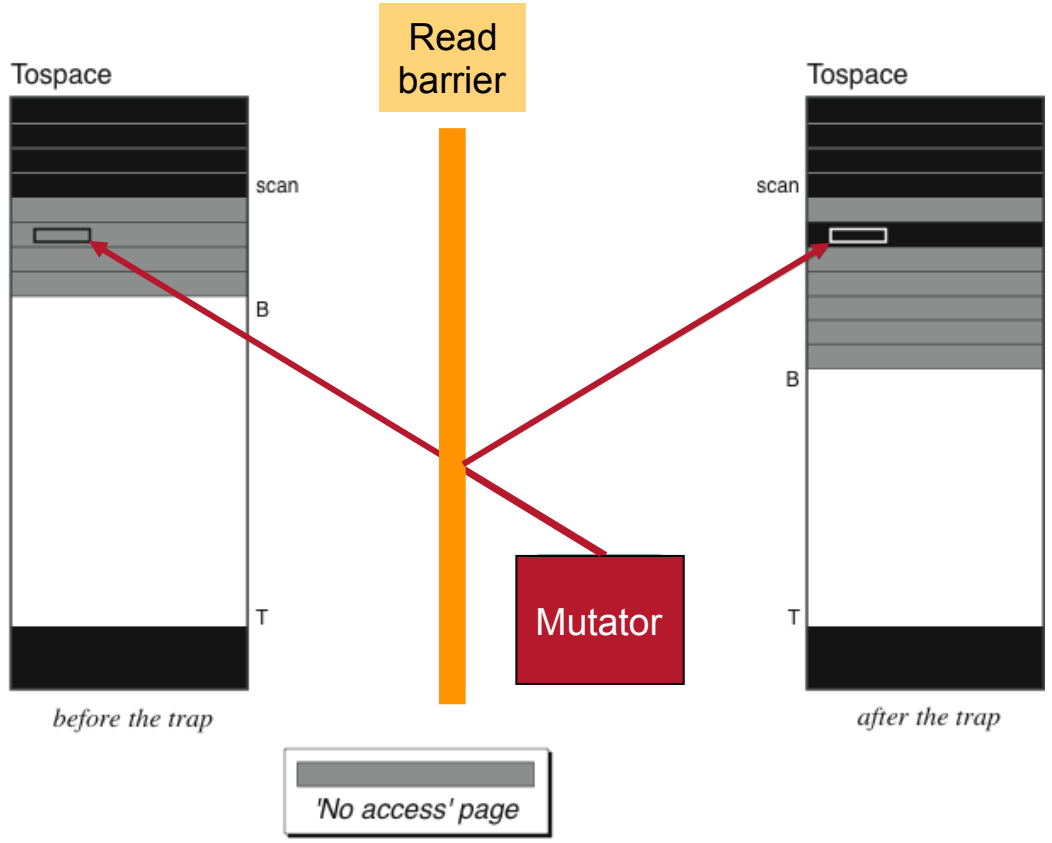
Pointer Fix-up During Mark

- When can a moved object be freed?
  - When there are no more pointers to it
- Mark phase updates pointers
  - Redirects forwarded pointers as it marks them
- Object moved in collection *n* can be freed:
  - At the end of mark phase of collection *n+1*



Memory Protection

By protecting a grey page, any access attempt by the mutator is trapped.



D. Concurrent sweeping

Sweeping concurrently (with mutator)

- Sweeper only modifies mark-bits and garbage.
- Both are invisible to mutator so no interference.
  - Except: it will have to add things to the free list, which the mutator will also be using (i.e., allocating from); can use locks

Parallel extension is also straightforward

- Chief issue is load balancing.

HotSpot Concurrent/Parallel GC

Concurrent GC (CMS)

+XX:UseConcMarkSweepGC | –XX:+CMSIncrementalMode

-XX:+UseCMSCompactAtFullCollection

- Serial: Mark roots
- Concurrent: Mark rest of heap (iteratively)
- Checkpoint: Stop the world; finish marking
- Concurrent Sweep: moves to the free list

Parallel GC

-XX:+UseParallelGC | -XX:+UseParNewGC |

-XX:ParallelGCThreads=<desired number> (num CPUs by default)

Can use the parallel young collector with CMS

See also http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf