



# Parallelism and Concurrency in Garbage Collection

David Detlefs

Java™ Technology Research Group

Sun Microsystems Laboratories (Burlington)

# Introduction

- ☞ The Java™ language is the first garbage-collected language to achieve wide-spread commercial success.
- ☞ Surprisingly (given origin) widespread adoption in server applications.
- ☞ Many of these have some real-time requirements.
- ☞ Java Technology Research Group has been trying to help Sun meet this challenge for five+ years.

# Outline

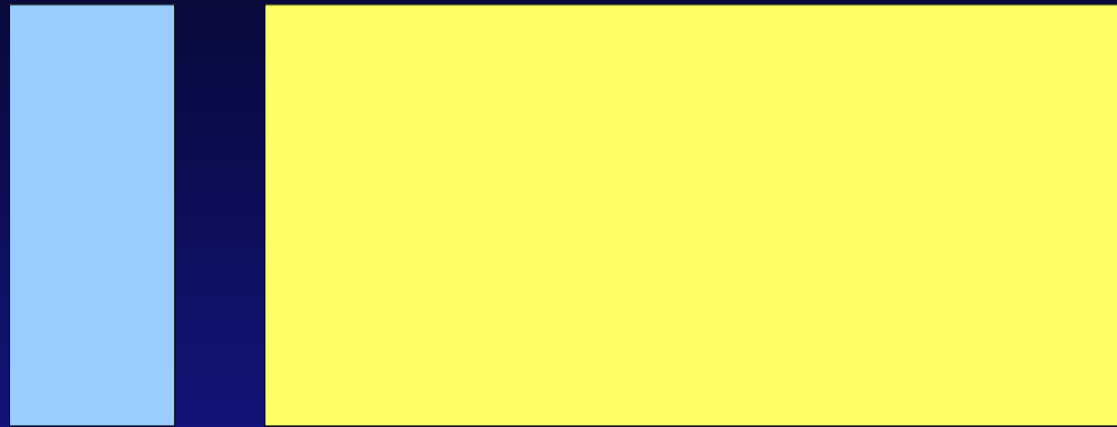
## ☞ Previous work:

- ☞ Generational mostly-concurrent collection.
- ☞ Parallel collection.

## ☞ More recent work:

- ☞ Combination: ParNew + CMS

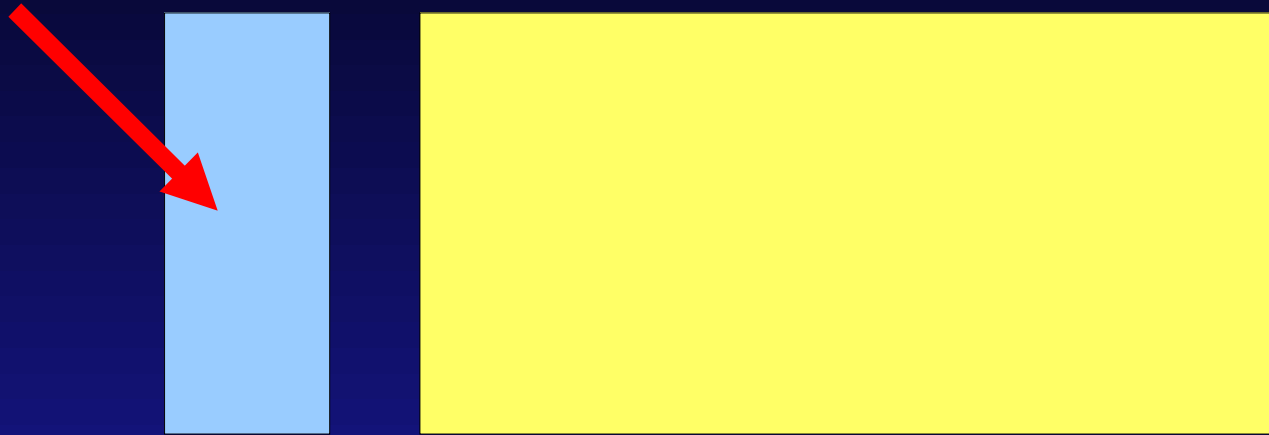
# Generational Collection



- Reduces GC pauses (i.e., latency).
- (Generally) increases throughput.

# Generational Collection

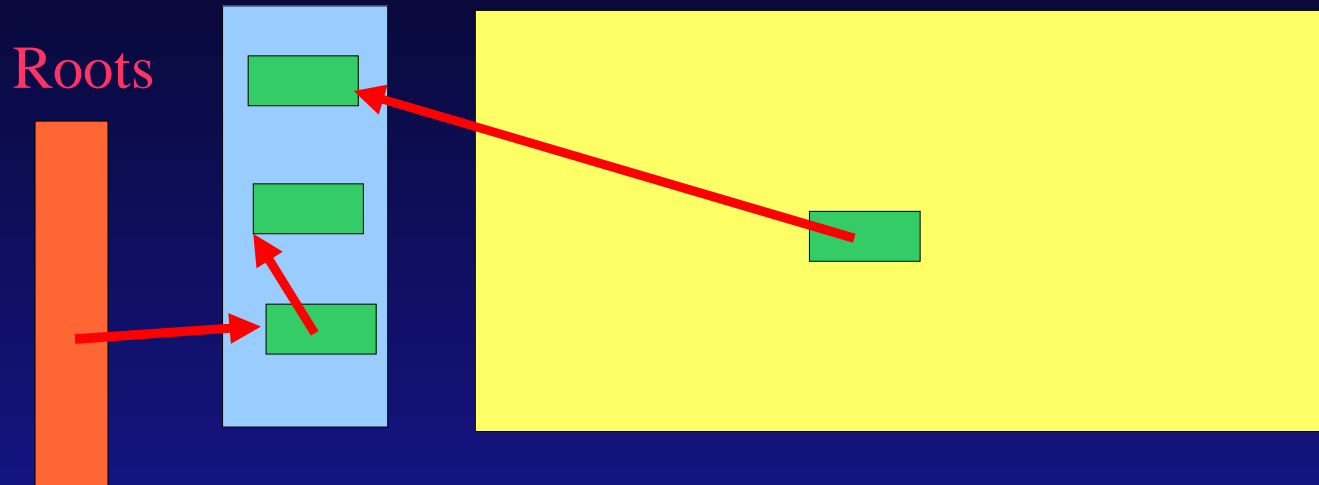
Allocation



- ➡ Reduces GC pauses (i.e., latency).
- ➡ (Generally) increases throughput.

# Generational Collection

## Young-gen collection



- 👉 Reduces GC pauses (i.e., latency).
- 👉 (Generally) increases throughput.

# Latency Problem

- ☞ Must eventually collect large (GByte+) old generations...
- ☞ ...without interrupting the program more than the (100 ms) limit...
- ☞ ...and scale with threads, processors, heap size, too!

# Concurrent Mark-and-Sweep

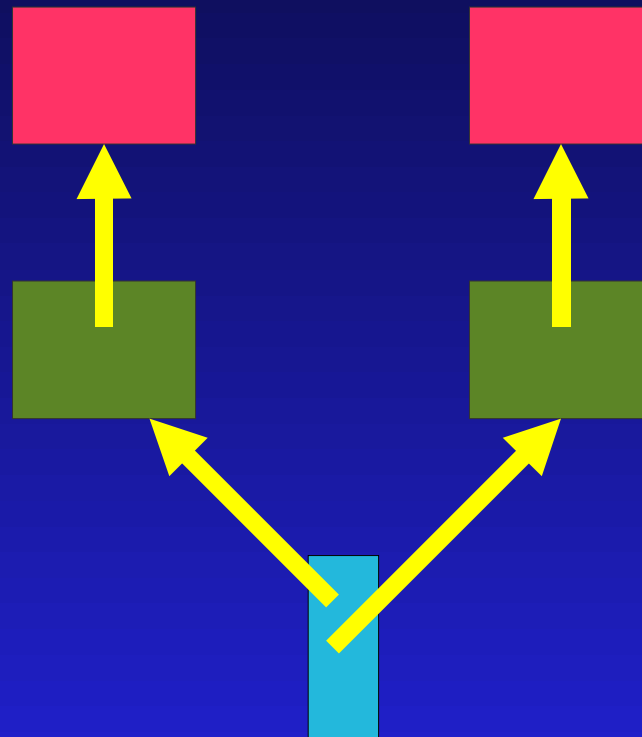
- Devote a processor to doing old-gen collection.
- (Almost entirely) concurrently with the mutator.



# Mostly-Concurrent M-S

(Boehm, Demers, Shenker)

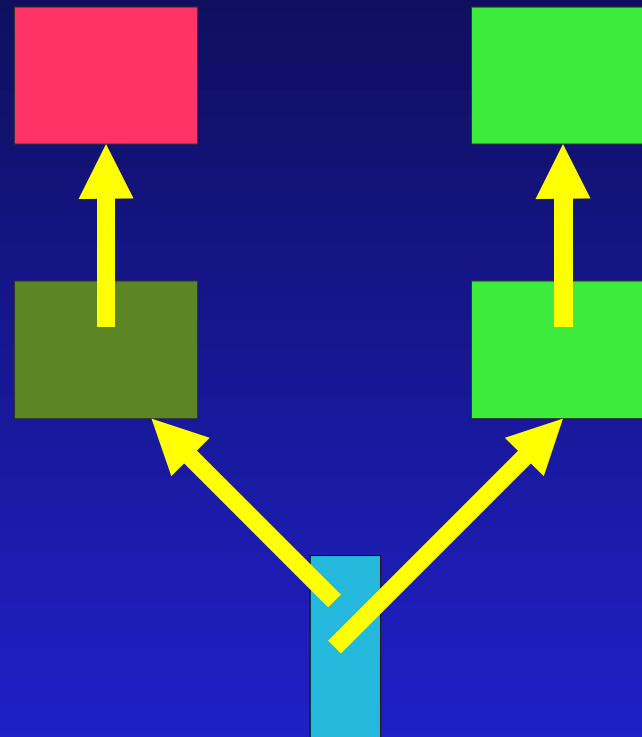
- Stop program, mark from roots.
- Restart program, mark concurrently.
- Problem:



# Mostly-Concurrent M-S

(Boehm, Demers, Shenker)

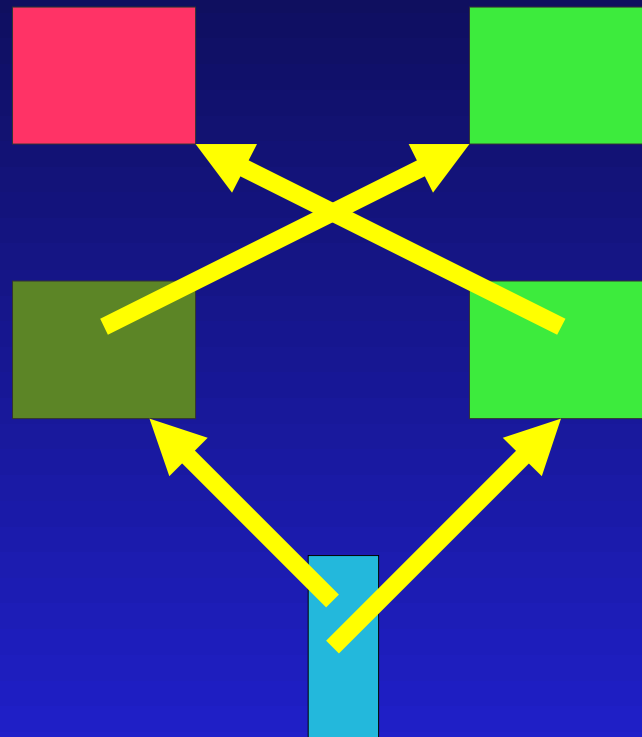
☞ Collector does some marking.



# Mostly-Concurrent M-S

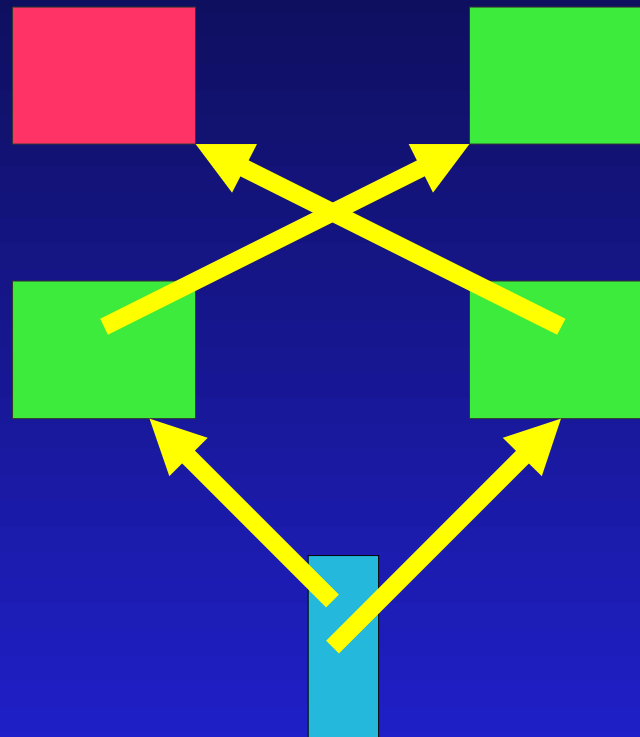
(Boehm, Demers, Shenker)

☞ Mutator updates some pointers.



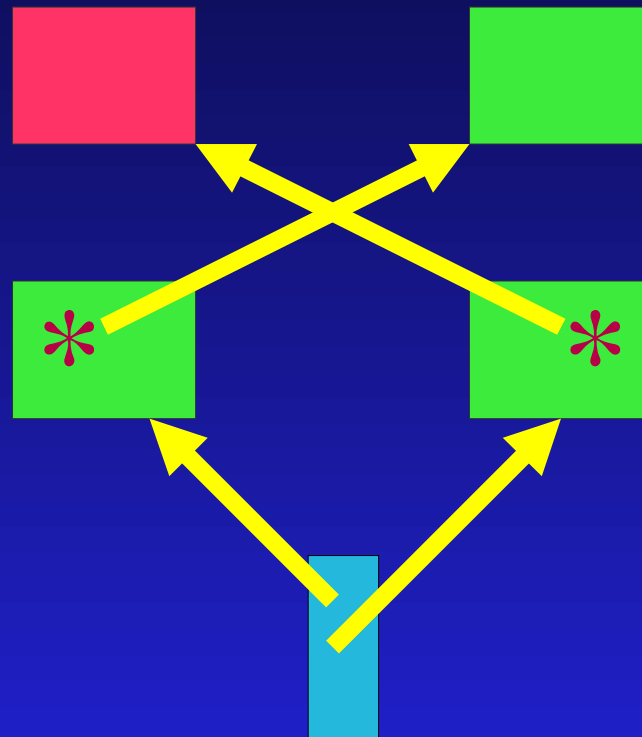
# Mostly-Concurrent M-S (Boehm, Demers, Shenker)

- Collector continues marking.
- At end, reachable object is not marked.



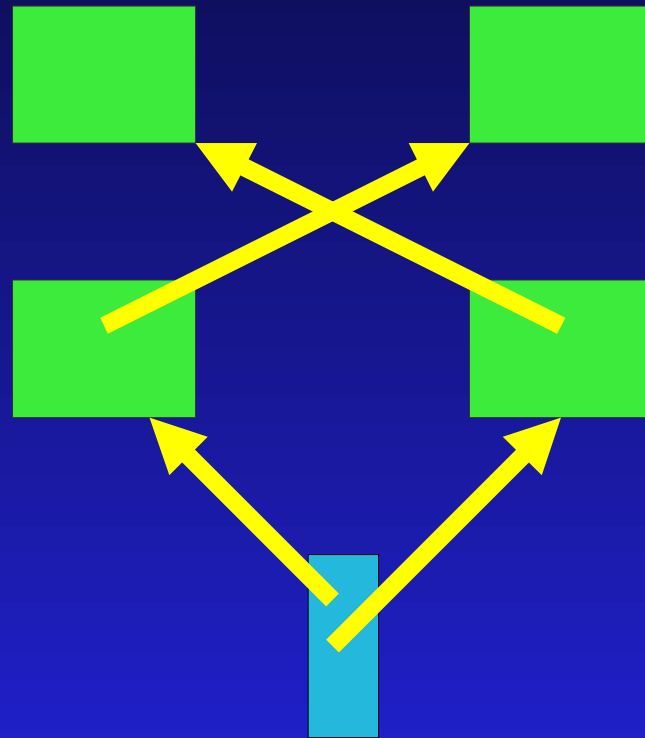
# Handling Concurrent Modification

- Mutator pointer writes marked objects as *dirty*.



# Handling Concurrent Modification

- When concurrent marking completes...
- Stop program, *remark* from
  - roots (again)
  - marked, dirty objects.
- Remark pauses short if concurrent updates rare.



# Concurrent Precleaning

- ☞ Some programs are not well-behaved:
  - ☞ high pointer mutation rate
  - ☞ remark phase is long.
- ☞ However: may remark (mostly) concurrently!
  - ☞ Objs may be dirtied after, so must still remark with world stopped. But many fewer dirty objs.
  - ☞ Concurrency issue: write barrier must do ptr store before barrier. (Or else:
    - `do-barrier (dirty obj); preclean; do store`
    - miss the update.)

# Rest of Collection

- Concurrent sweeping not too difficult.
- No compaction phase: *non-moving collector*
  - Pointers must be updated globally when an object moves.
  - Hard to do global operations concurrently.
  - (But we have some ideas!)
  - Therefore, free-list-based allocation.

# Our Innovations

(Detlefs, Printezis, ISMM'00)

- Use of CMS as old generation in two-generation system.
- Fast linear allocation in young generation.
- Slower free-list allocation only during promotion (relatively rare.)
- One (card-table) write barrier serves to track both:
  - old-to-young pointers for generational GC.
  - modified objects for CMS.

# Typical Conc-MS Results

- ☞ For a synthetic benchmark (old result).
- ☞ 250 MB live data in heap; processor available for GC.

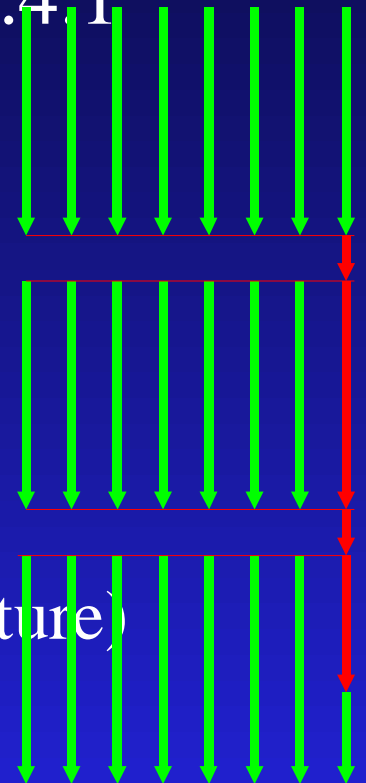
	Elapsed sec	Max heap (MByte)	young-gen pause times			old-gen pause times		
			avg ms	max ms	total sec	avg ms	max ms	total sec
MSC	370	415	21	38	59.6	6944	10368	34.7
Conc-MS	356	498	31	145	91.1	41	105	0.6

☞ There exist > 1 telco *call processing* applications using this "in anger." Typical constraint: 500 ms.

☞ These applications are infeasible in Java

# CMS: Implementation Status

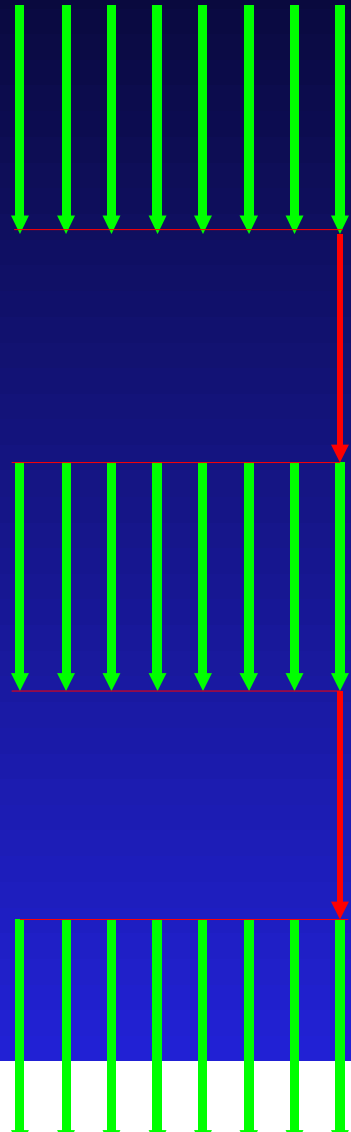
- Available to select customers in 1.2 systems for several years.
- Now implemented in HotSpot™ VM 1.4.1
  - Y. S. "Ramki" Ramakrishna, Jon Masamitsu, John Coomes, Ross Knippel
- Further work:
  - "Incremental" CMS
  - Parallel remark (in progress)
  - Mostly-concurrent partial compaction (future)



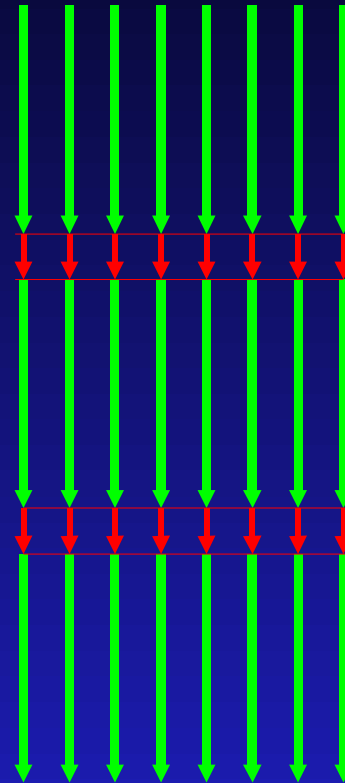
# Parallel Collection

(Flood, Detlefs, Shavit, Zhang, USENIX JVM'01)

Bad!



Good!



# Parallelization techniques

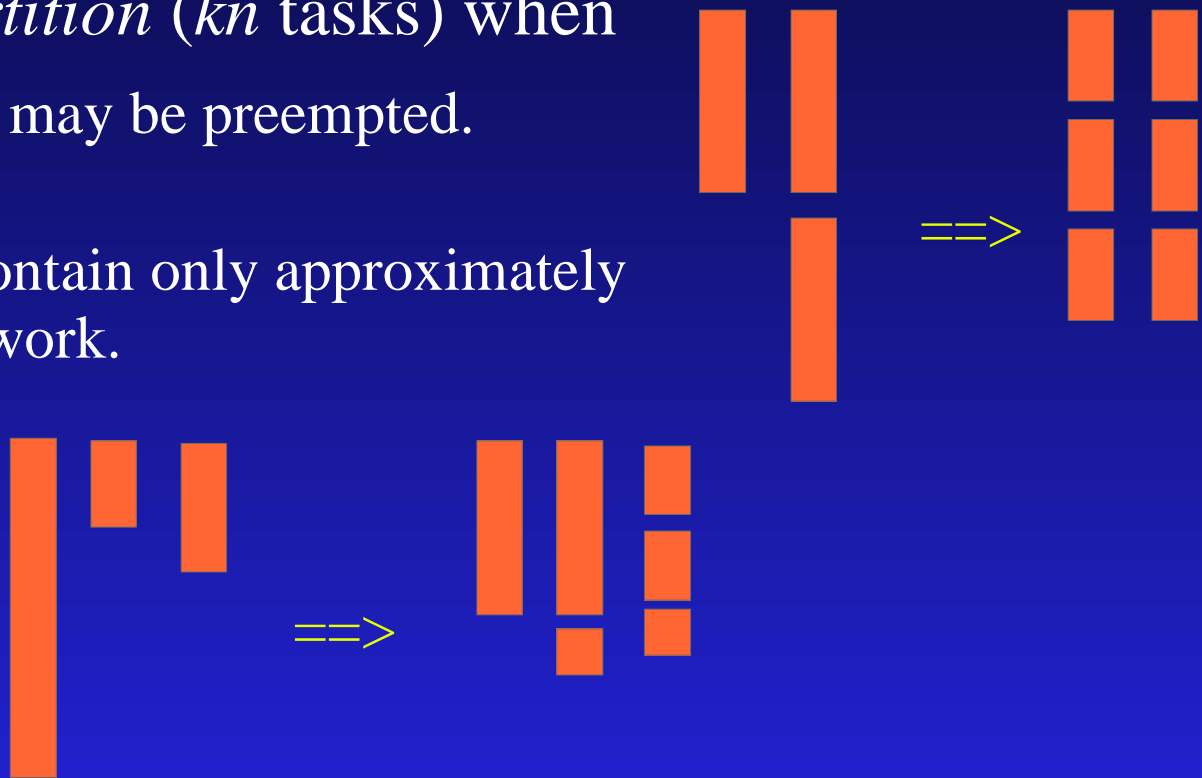
## ☞ Static partition.

☞ If  $n$  processors, divide work into  $n$  equal tasks.

☞ *Overpartition* ( $kn$  tasks) when

☞ threads may be preempted.

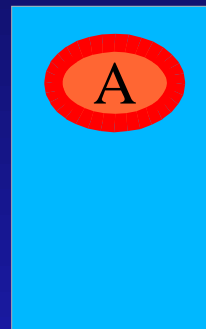
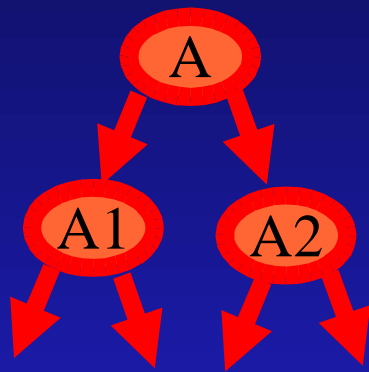
☞ tasks contain only approximately equal work.



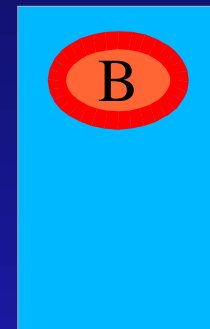
# Parallelization techniques

☞ Randomized work stealing.

- ☞ When tasks may create new tasks, unpredictably.
- ☞ Threads without tasks steal tasks from other threads.



Thread 1

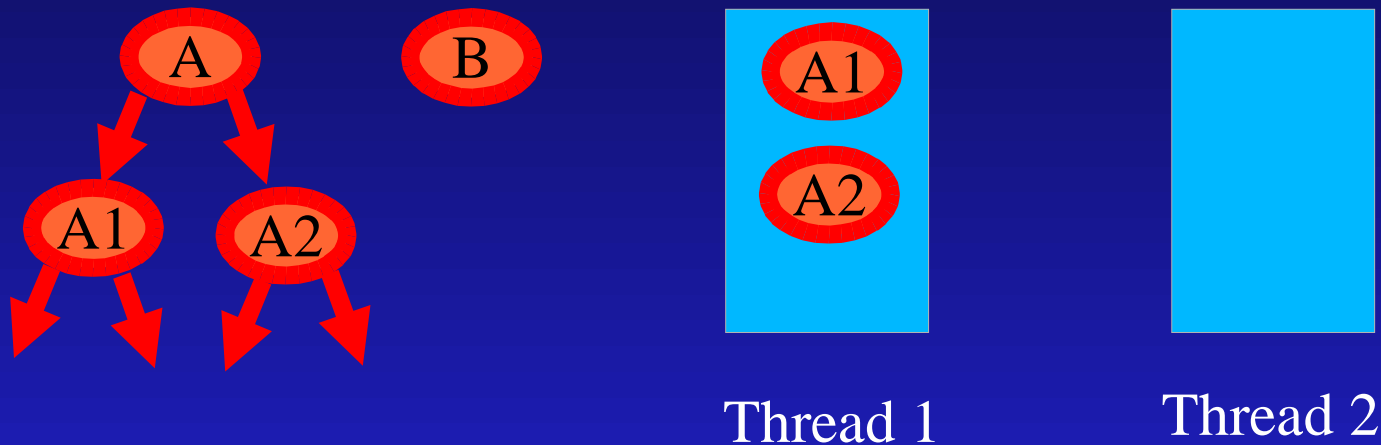


Thread 2

# Parallelization techniques

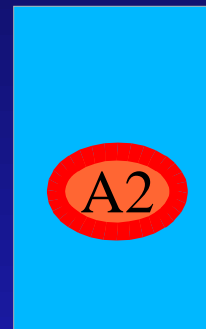
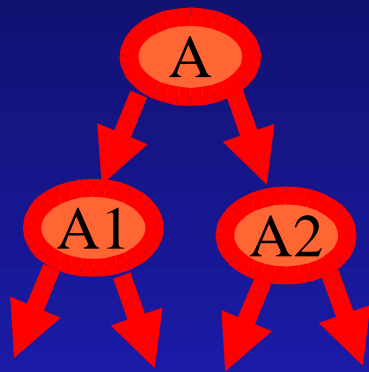
☞ Randomized work stealing.

- ☞ When tasks may create new tasks, unpredictably.
- ☞ Threads without tasks steal tasks from other threads.

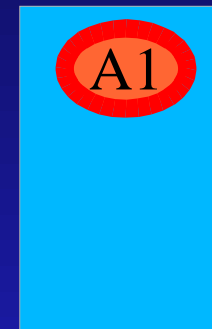


# Parallelization techniques

- Randomized work stealing (MIT Cilk project)
  - When tasks may create new tasks, unpredictably.
  - Threads without tasks steal tasks from other threads.

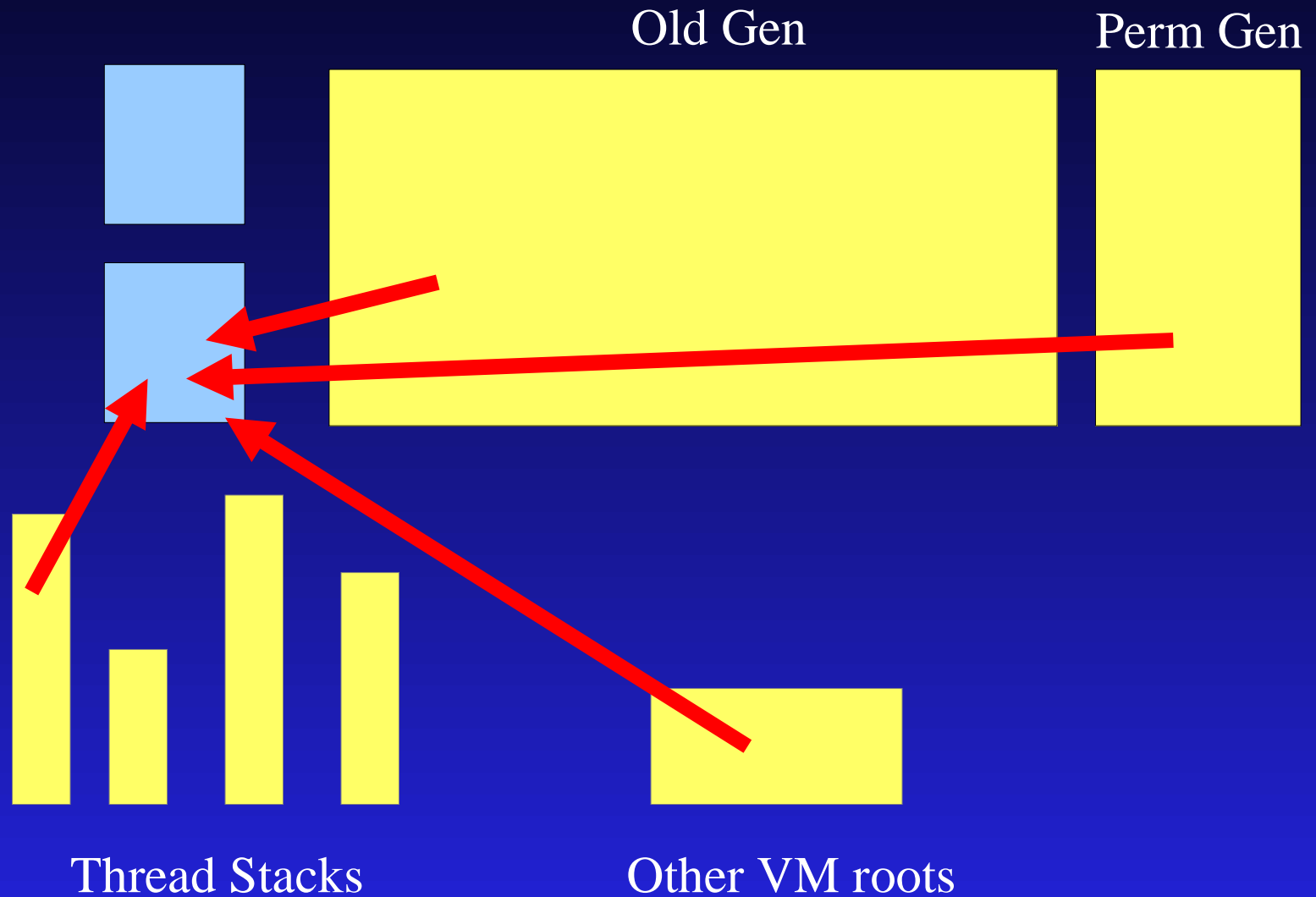


Thread 1

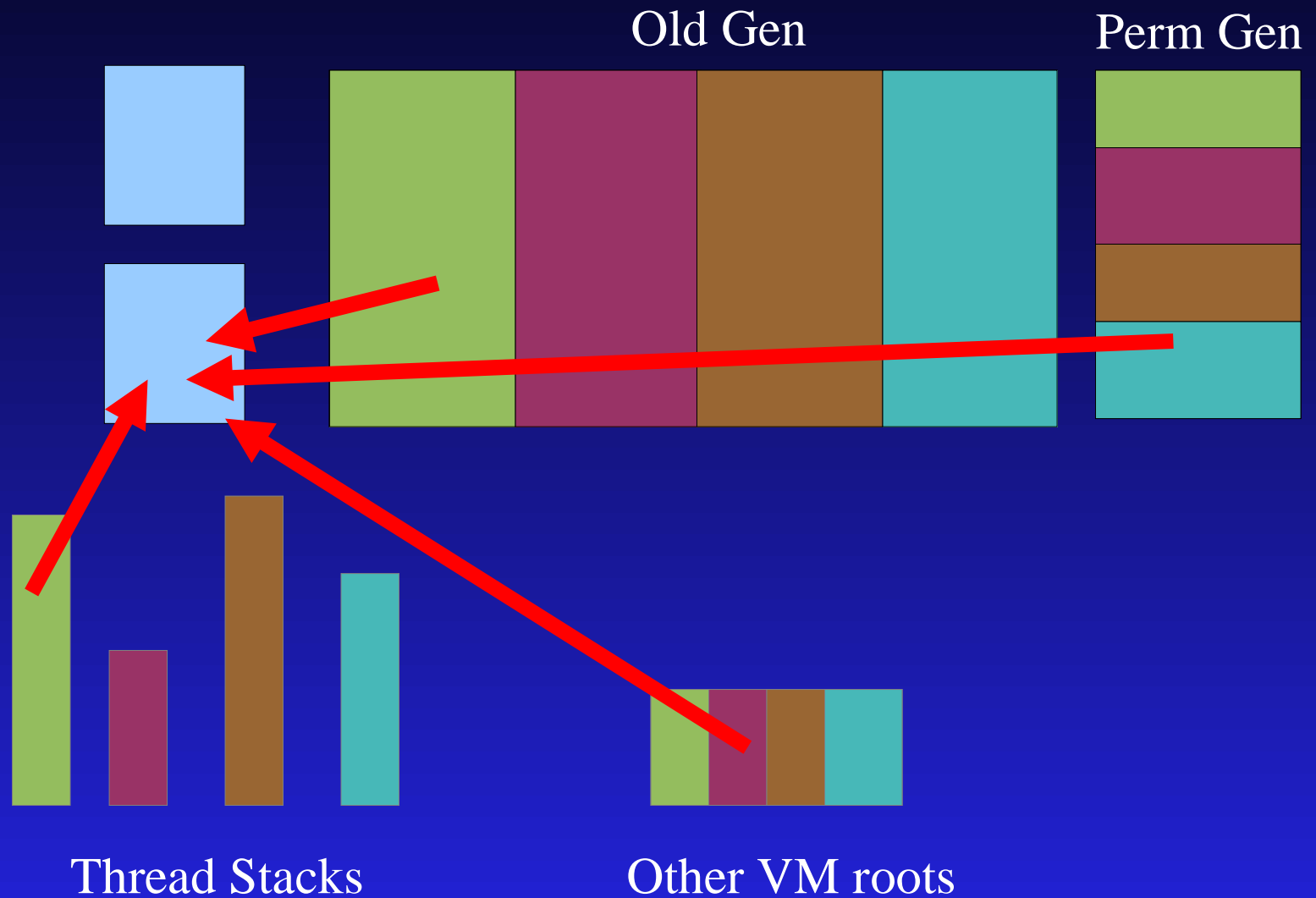


Thread 2

# Applying Techniques to Parallel Generational Collection: Root Partitioning

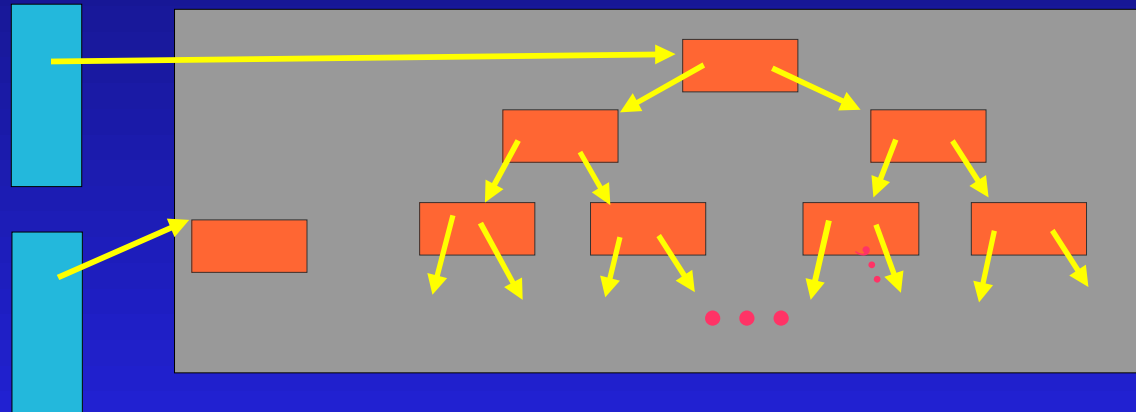


# Applying Techniques to Parallel Generational Collection: Root Partitioning



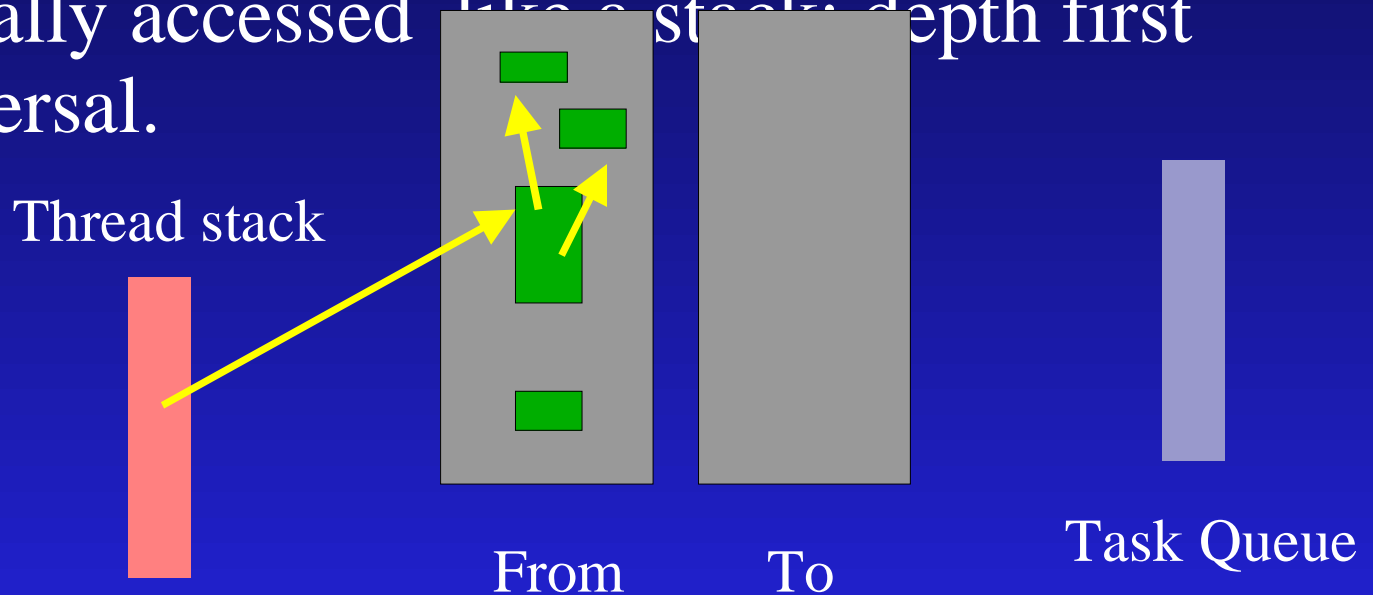
# Applying techniques to Parallel Collection

- Object graph traversal uses
  - Static partition of the root sets for parallel root scanning.
  - Then work-stealing to complete scanning of reachable objects.



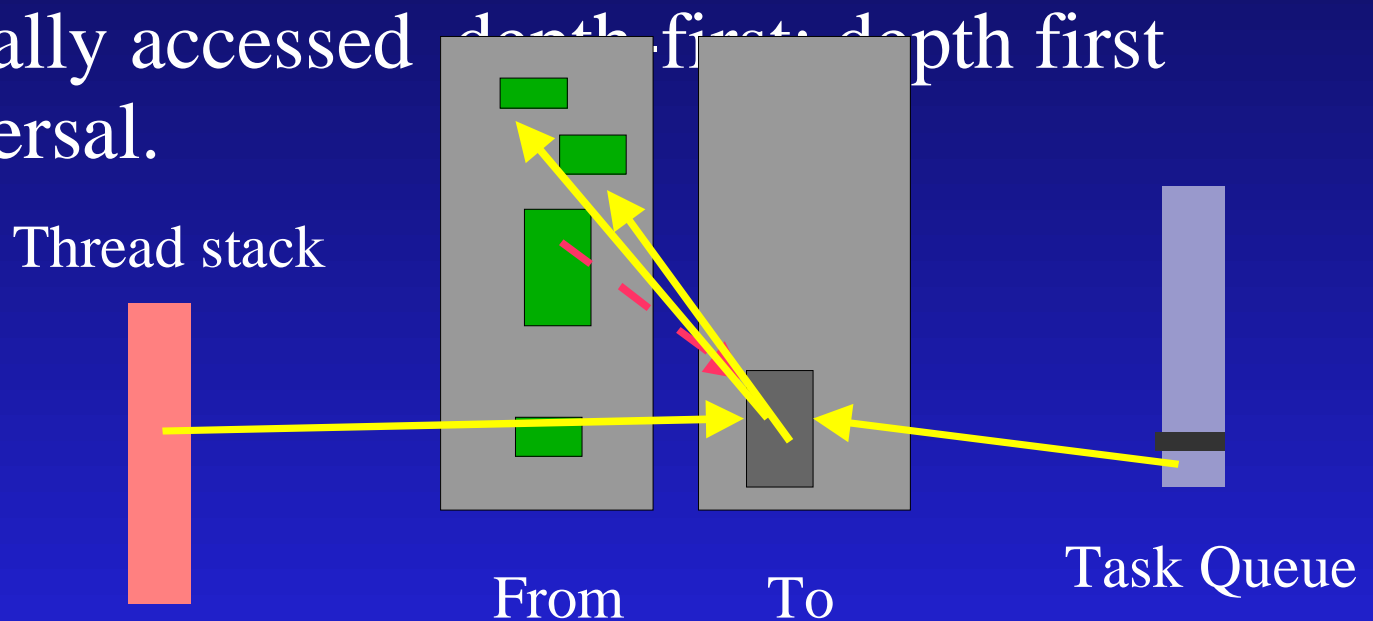
# Parallel Copying Collection

- Uses per-thread "1.5-end queues" (Arora, Blumofe, *et al.*) to track "gray" objects.
- Push and pop locally on one end; steal from the other.
- Locally accessed like a stack; depth first traversal.



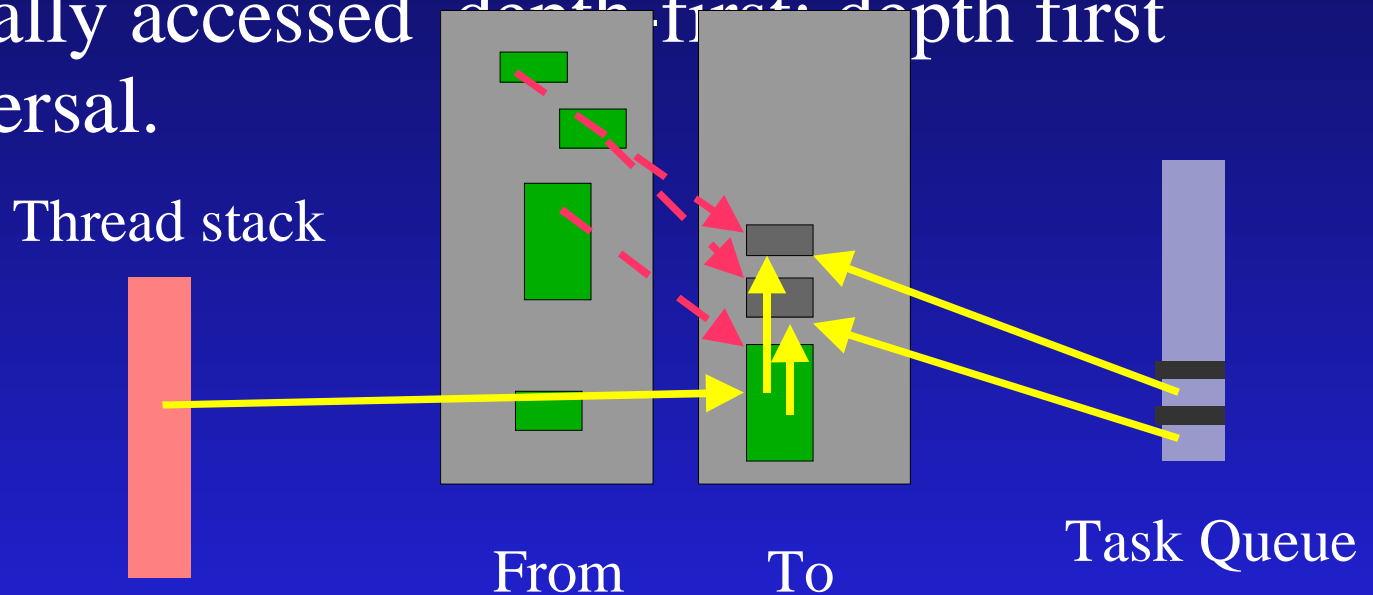
# Parallel Copying Collection

- Uses per-thread "1.5-end queues" (Arora, Blumofe, *et al.*) to track "gray" objects.
- Push and pop locally on one end; steal from the other.
- Locally accessed depth-first, depth first traversal.



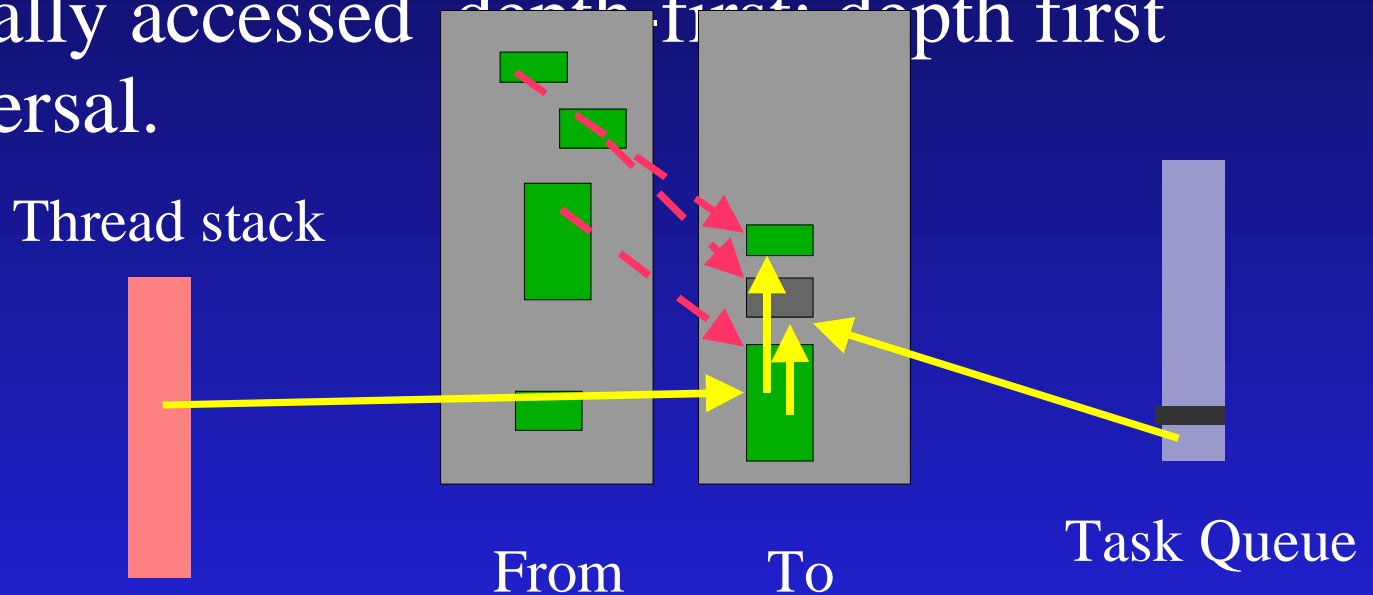
# Parallel Copying Collection

- Uses per-thread "1.5-end queues" (Arora, Blumofe, *et al.*) to track "gray" objects.
- Push and pop locally on one end; steal from the other.
- Locally accessed depth-first; depth first traversal.



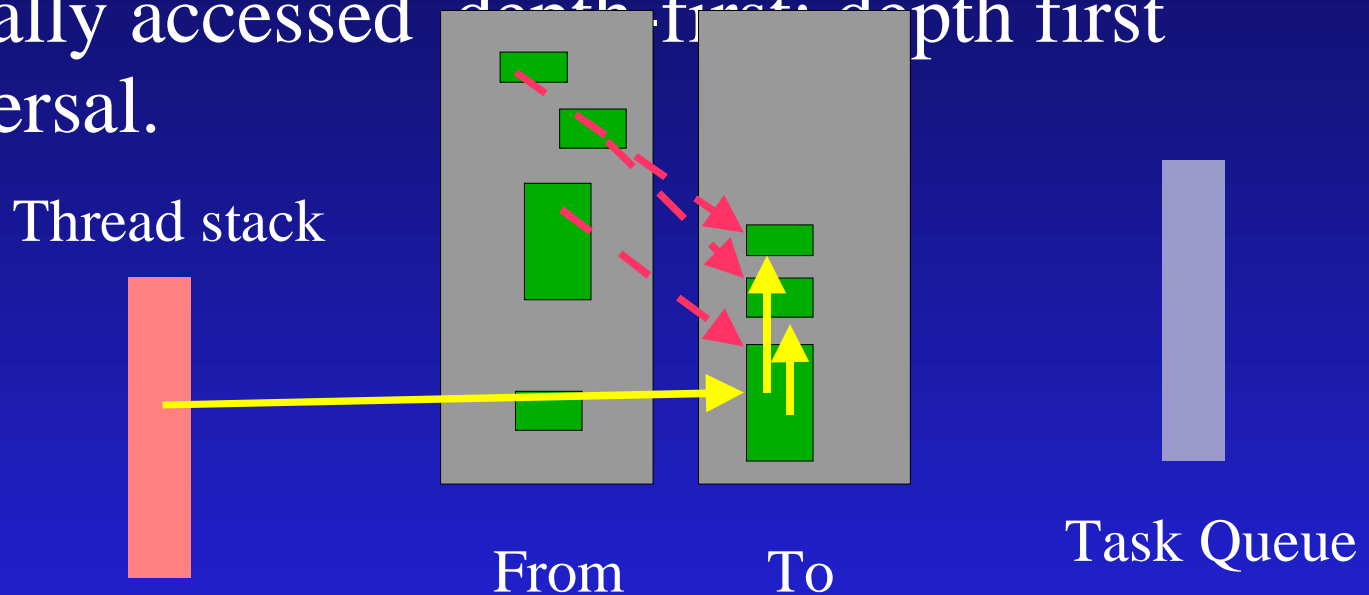
# Parallel Copying Collection

- Uses per-thread "1.5-end queues" (Arora, Blumofe, *et al.*) to track "gray" objects.
- Push and pop locally on one end; steal from the other.
- Locally accessed depth-first, depth first traversal.



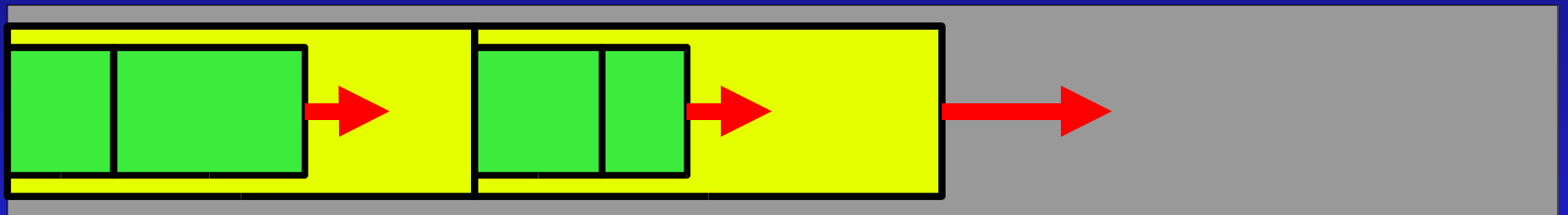
# Parallel Copying Collection

- Uses per-thread "1.5-end queues" (Arora, Blumofe, *et al.*) to track "gray" objects.
- Push and pop locally on one end; steal from the other.
- Locally accessed depth-first, depth first traversal.



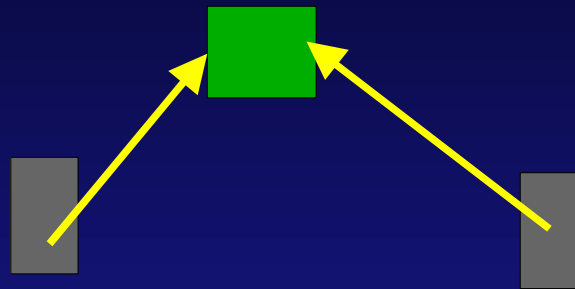
# Parallel Copying Collection

- ☞ *local allocation buffers* for to-space allocation.
  - ☞ Allocate a medium chunk from global allocator (lock or CAS).
  - ☞ Allocate from LAB's without synchronization.
  - ☞ Simplifies situation on next slide...



# Parallel Copying Collection

☞ One race condition: who gets to copy an object?



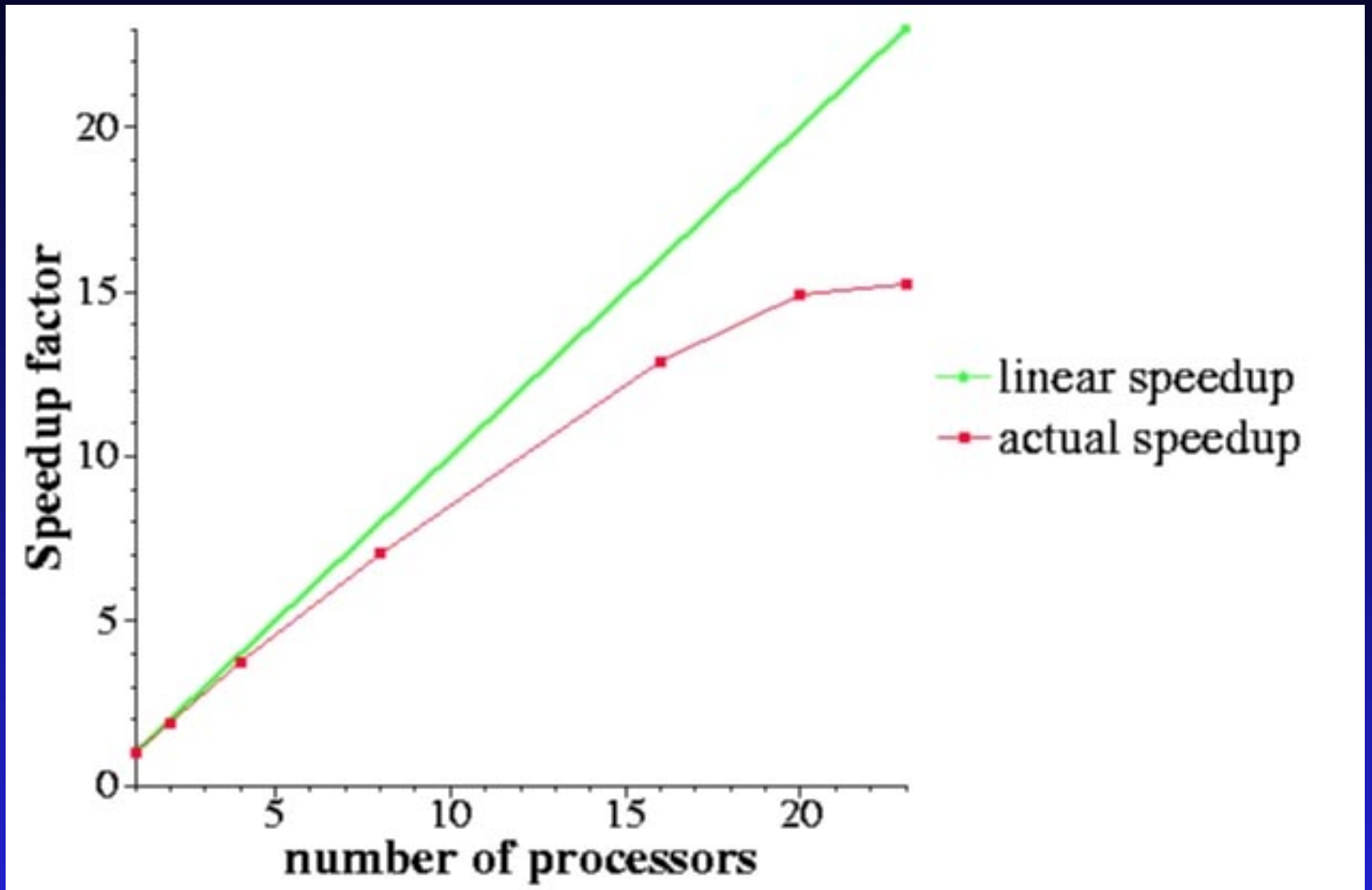
☞ Resolve by inserting forwarding pointers with CAS:

☞ Allocate speculative to-space address in LAB.

☞ Try to insert into header with CAS.

☞ If lose the race: undo LAB allocation (easy, since local).

# Performance (SPECjbb)



# Parallel Young-gen Collection: Status

- Available in HS 1.4.1 in two flavors:
  - ParNew (me, Christine Flood, Ross Knippel, Joe Provino)
  - Parallel Scavenge (James McIlree, Peter Kessler)
- These are largely algorithmically isomorphic, and have similar performance.
- Adventurous customers are starting to use it:
  - "The best VM advance in several years." (Paul Ciciora, CBOE)

# Parallel Young + CMS Old

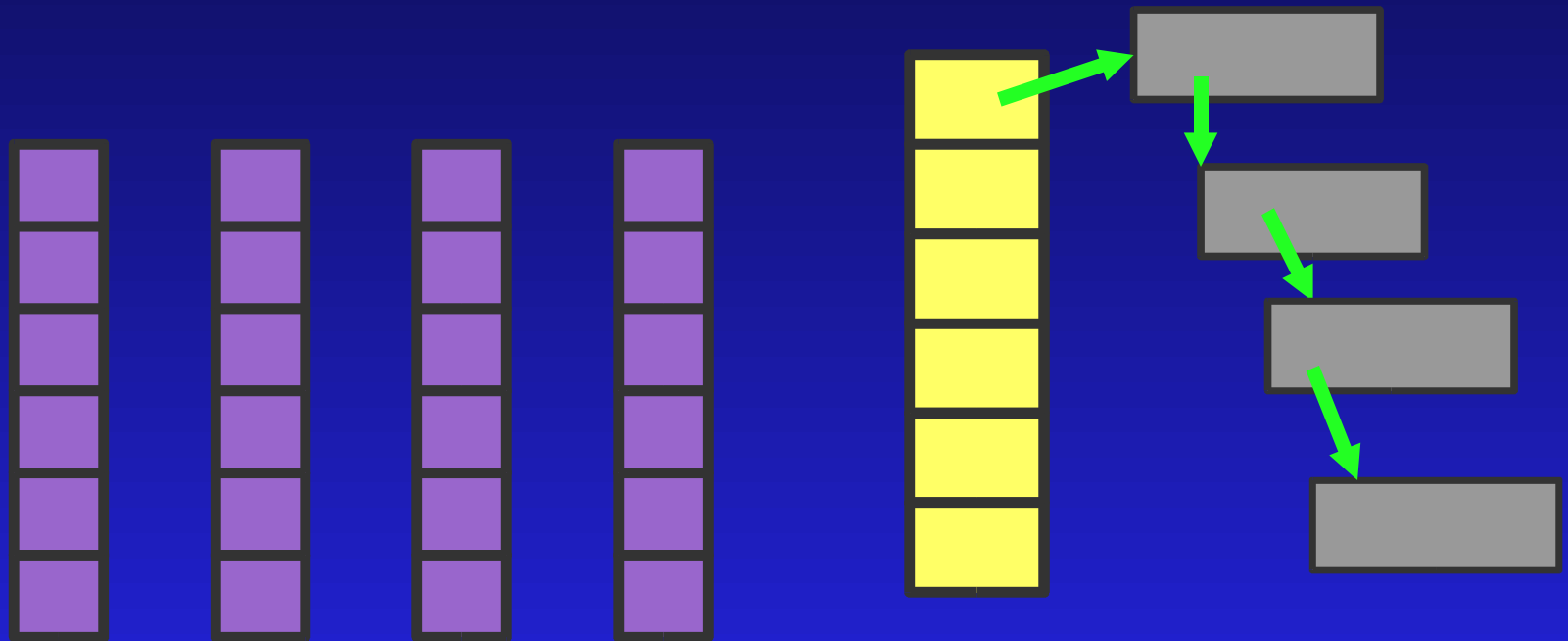
- The promise: high throughput, low latency.
- "Parallel" old gen collection!
- Should be easy: *Generational Framework* tries to make generations "pluggable".
- Largely it is, with some interesting exceptions we explore here.

# Par New + CMS: Problems

- Parallel free-list allocation.
- Parallel card scanning and promotion.
- Avoiding promotion undo.

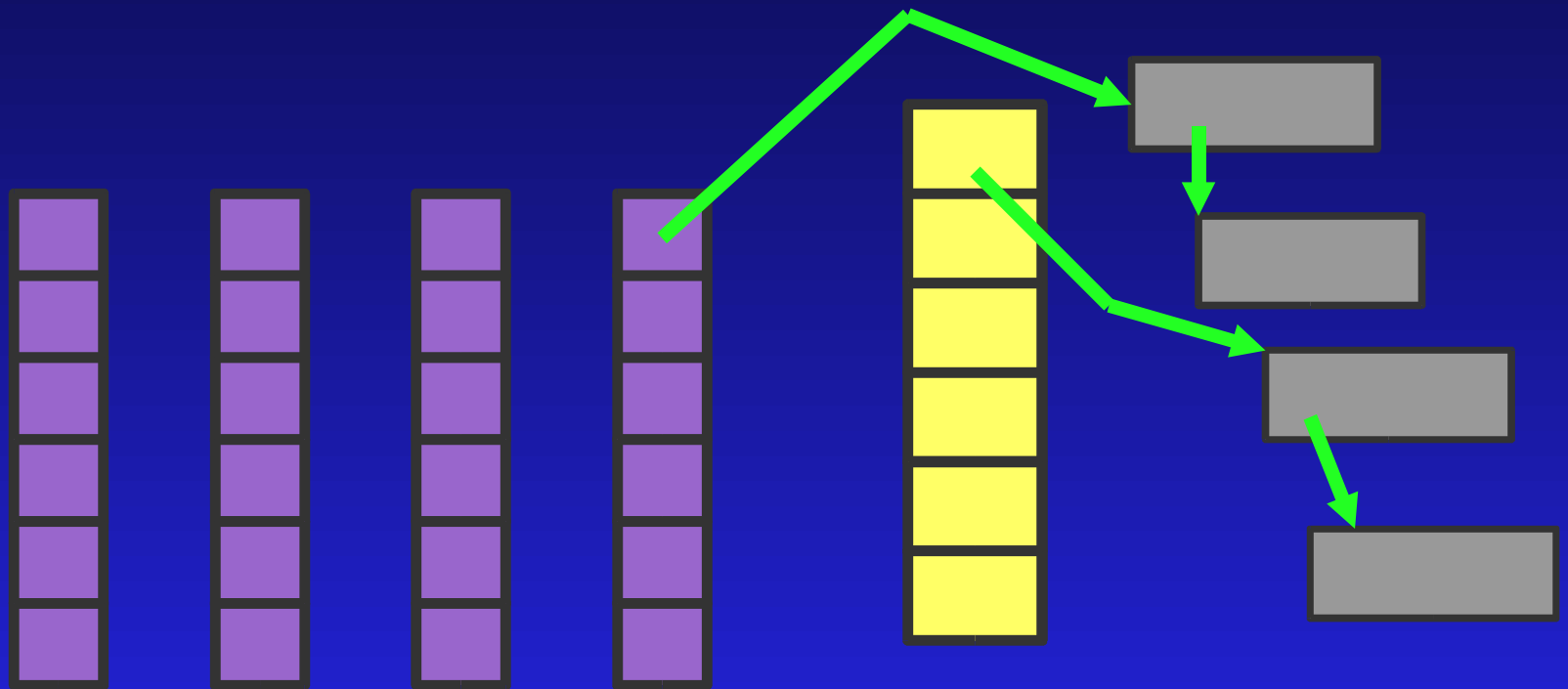
# Parallel Free List Allocation

- 👉 Big objects: locked allocation from shared tree structure.
- 👉 Small objects: "Thread-local free lists"



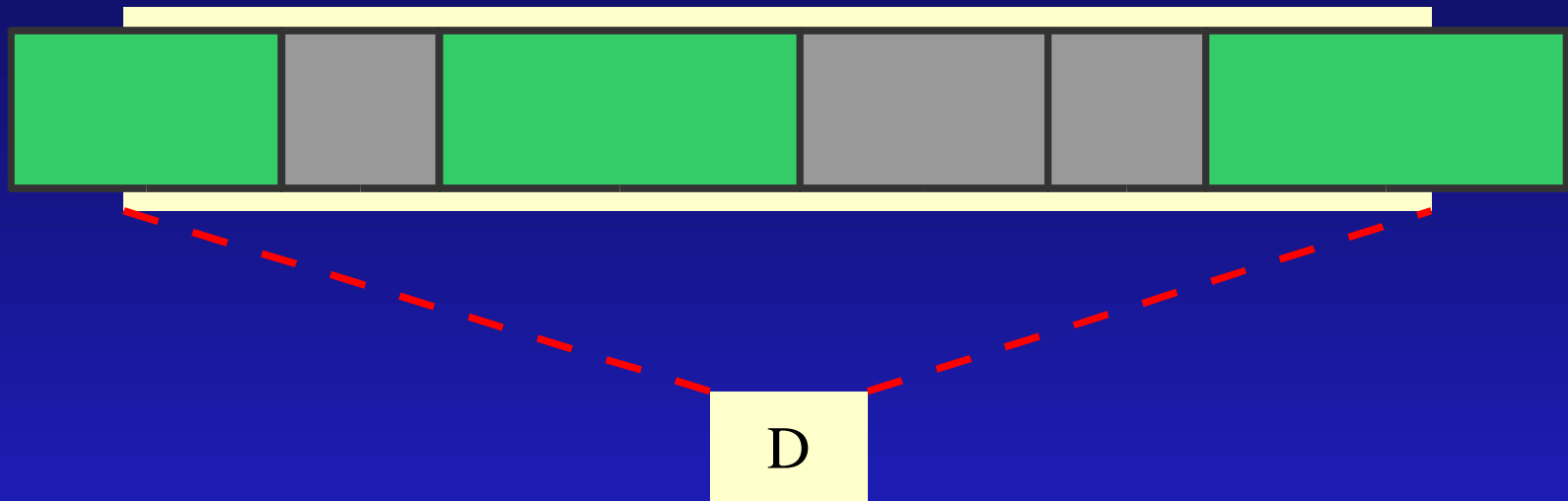
# Parallel Free List Allocation

- 👉 Big objects: locked allocation from shared tree structure.
- 👉 Small objects: "Thread-local free lists"



# Parallel Card Scanning and Promotion

☞ Consider a dirty old-generation card:



# Parallel Card Scanning and Promotion

☞ Consider a dirty old-generation card:

Thread A: scanning

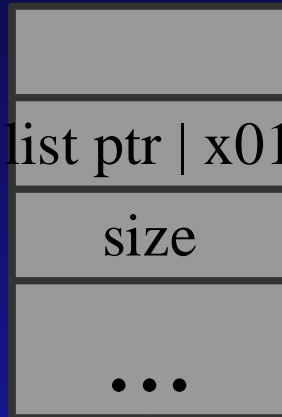


D

Thread B: promoting

# CMS Block Formats

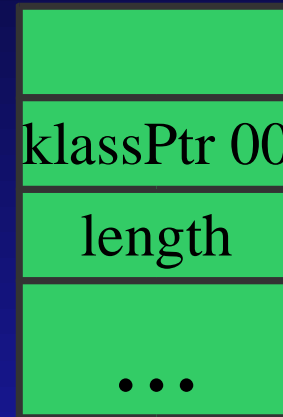
☞ Free blocks



Object



Array

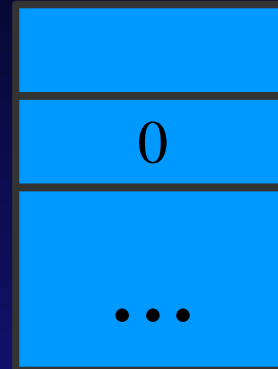


☞ How do we change from free block to array?

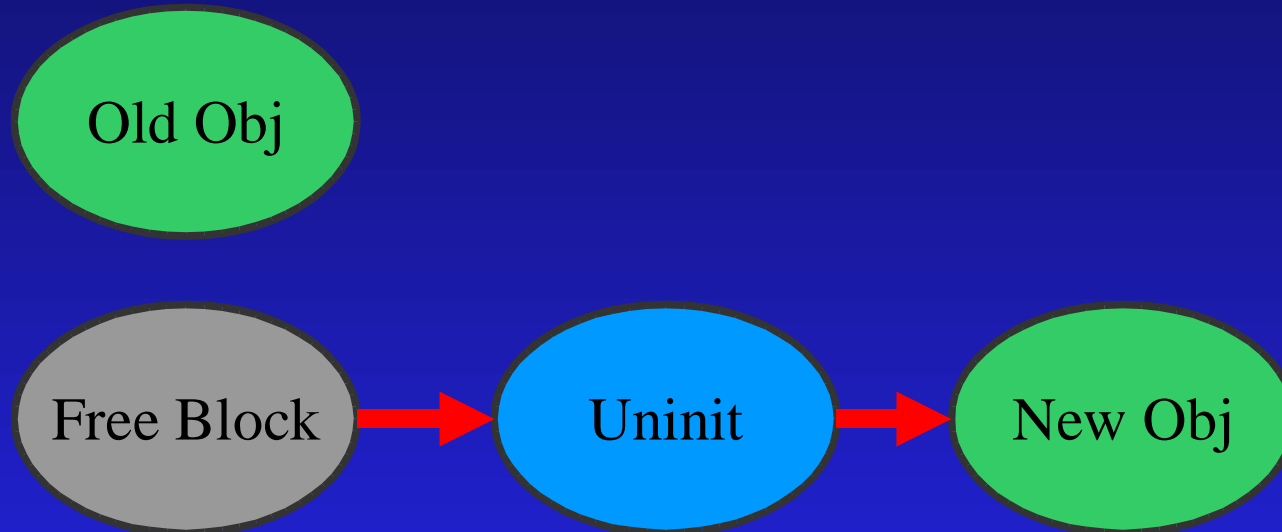
☞ How do we reliably find the block size?

# Parallel Card Scanning and Promotion: Solution

➡ New state: *uninit*



➡ Allowed state transitions:



# Parallel Card Scanning and Promotion: Solution

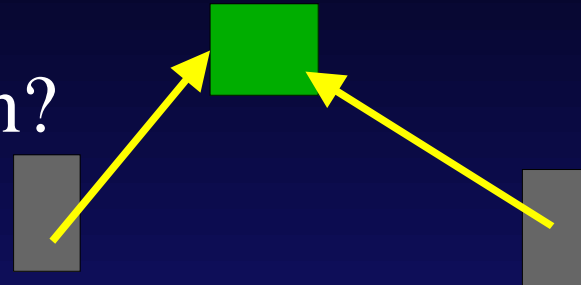
```
size_t block_size(blk) {
    while (true) {
        w2 = blk.word2;
        if (is_free_block(w2)) {
            sz = blk.word3;
            if (is_free_block(blk.word2)) return
sz;

            else continue;
        } else if (is_obj(w2)) {
            return obj_size(blk);
        }
        // Otherwise, w2 == 0; spin-wait.
    }
}
```

# Avoiding Promotion Undo

☞ Remember this situation?

☞ Before: if you lose the race, undo the allocation.



☞ Now: obj->free\_block transition breaks "block\_size".

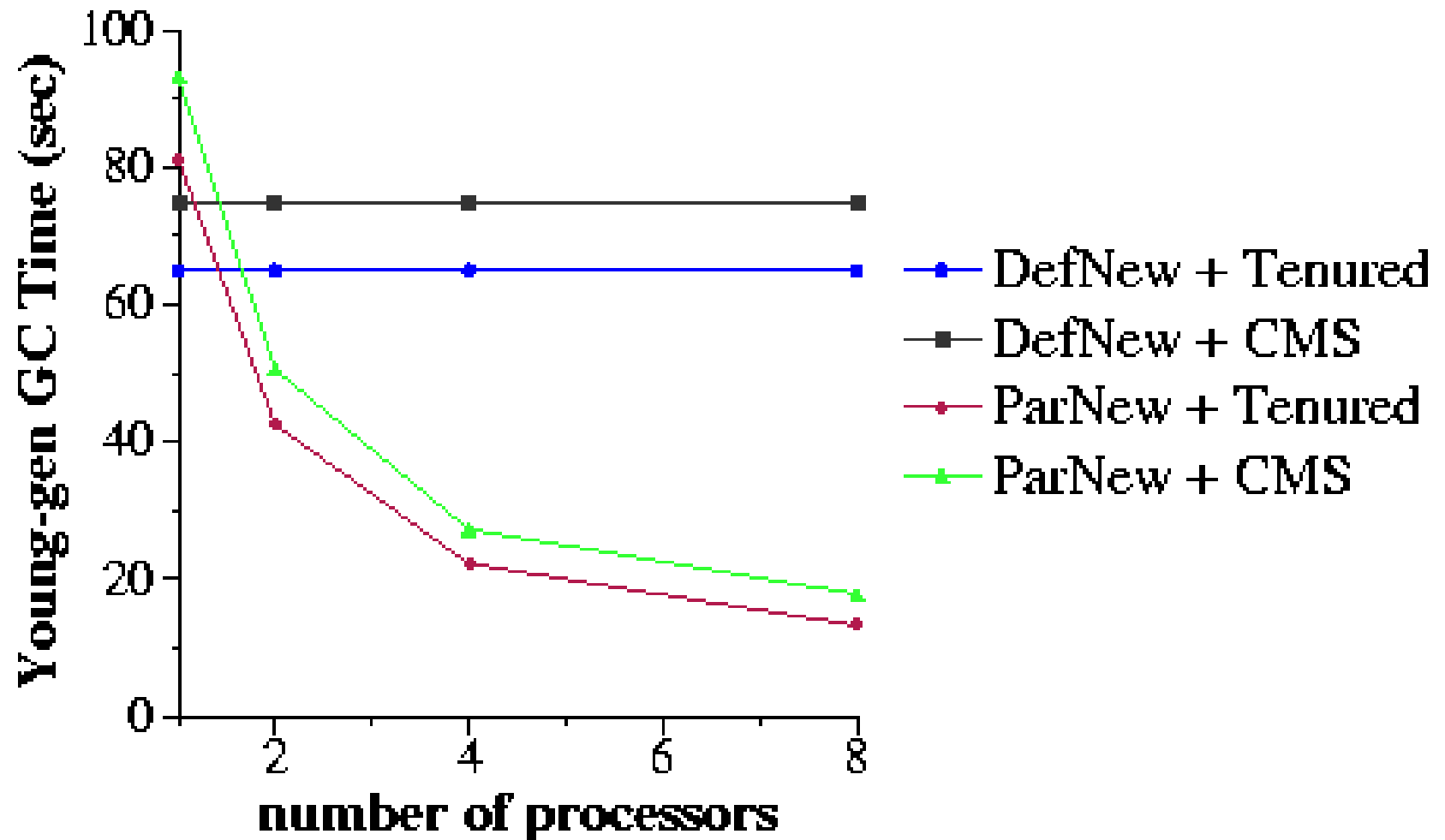
☞ So we avoid promotion undo:

☞ Insert special *claimed* forwarding pointer with CAS.

☞ If win, then do allocation, insert real forwarding pointer.

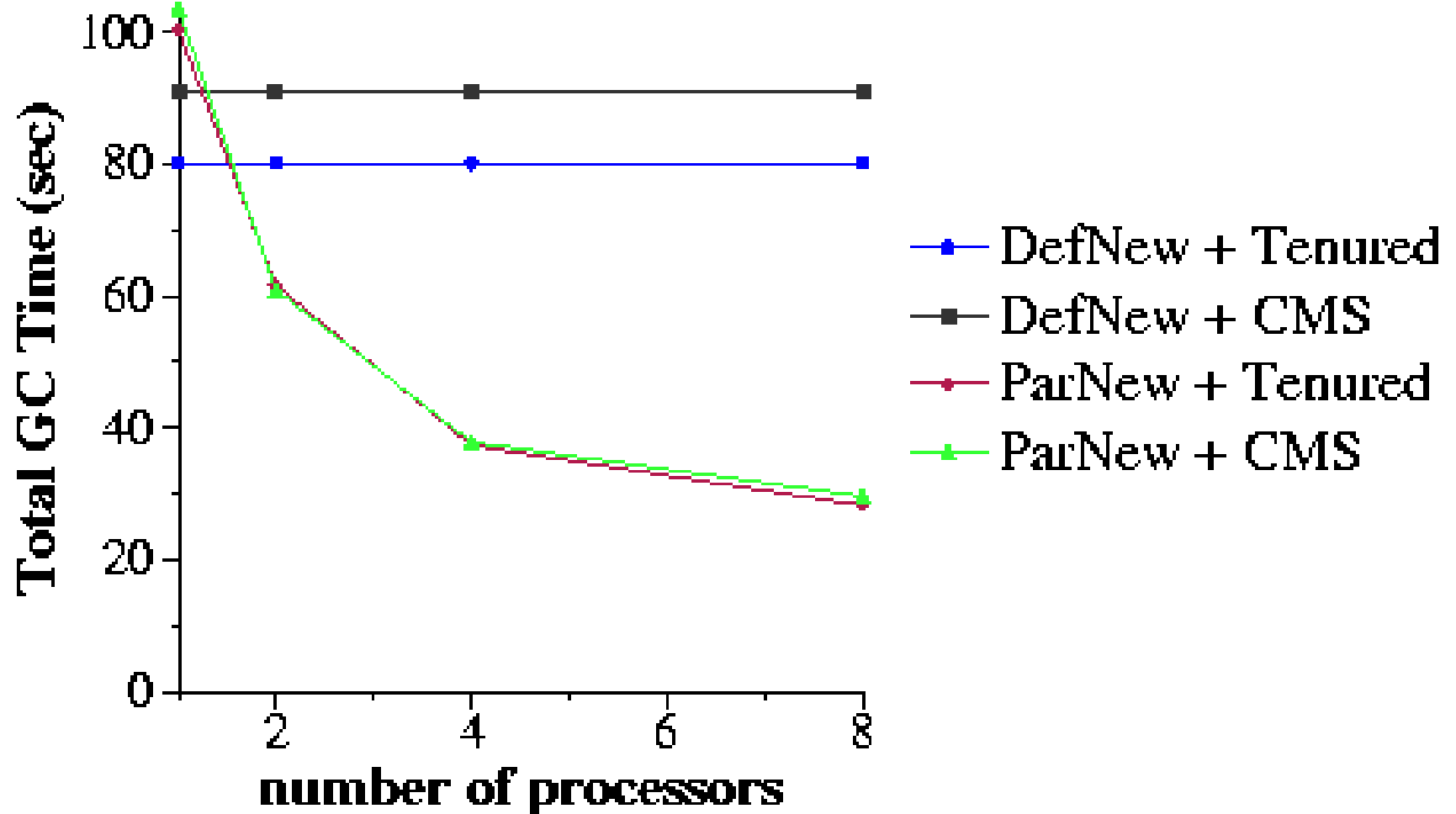
☞ Thread that observes *claimed* spins until real.

# ParNew+CMS: Performance



SPECjbb: Young-gen GC

# ParNew+CMS: Performance



SPECjbb: Total GC

# Conclusions

- ☞ We have good attacks on large-heap latency and scalability.
  - ☞ Concurrent collection.
  - ☞ Parallel collection.
- ☞ And the obvious combination:  
Par Young + Concurrent Old.
- ☞ An excellent resource:  
[http://developer.java.sun.com/developer/  
/  
technicalArticles/Programming/turbo/](http://developer.java.sun.com/developer/technicalArticles/Programming/turbo/)

# Future Directions

## ☞ CMS:

- ☞ Parallelize stop-world parts of CMS.
- ☞ > 1 CMS thread for large-scale machines.
- ☞ Mostly-concurrent partial compaction (Tony Printezis).
- ☞ "Garbage-First" GC (another talk...:-)

