Parallel GC

Terms

Parallel GC

- Mutator paused while GC runs
- Multiple GC threads
- Hopefully better throughput

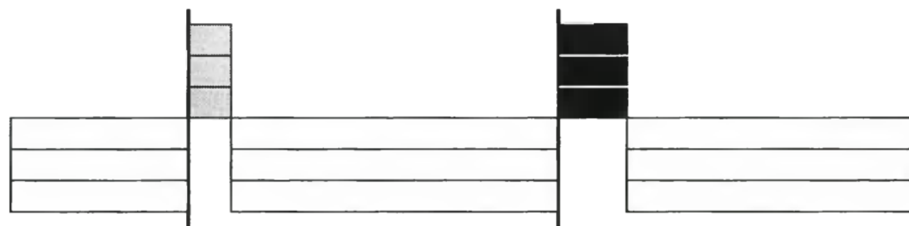
Concurrent GC

- Mutator and GC run concurrently
- One or more GC threads
- Hopefully less latency

time

(a) Stop-the-world collection, single thread

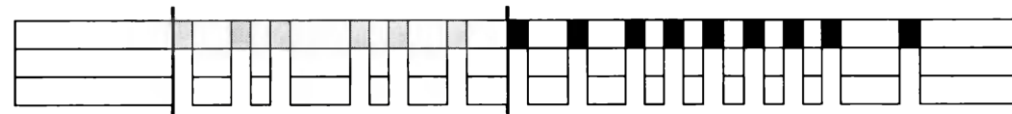
(b) Stop-the-world collection on multiprocessor, single collector thread



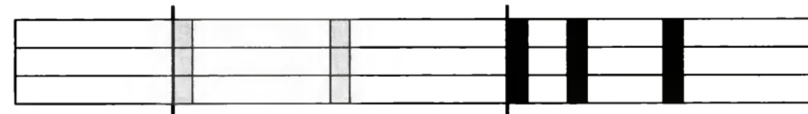
(c) Stop-the-world parallel collection

time

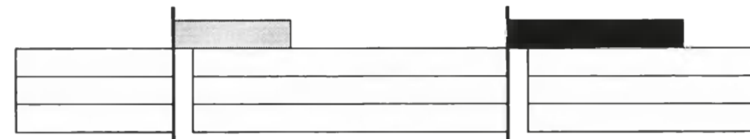
(a) Incremental uniprocessor collection



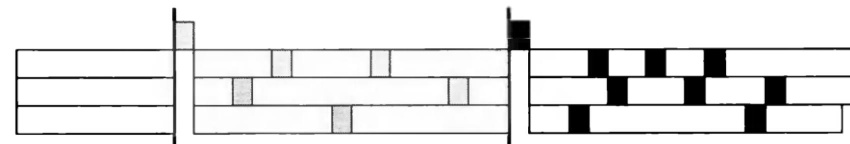
(b) Incremental multiprocessor collection



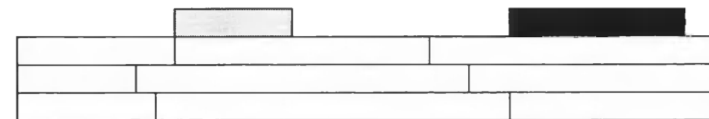
(c) Parallel incremental collection



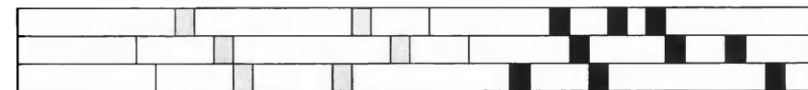
(d) Mostly-concurrent collection



(e) Mostly-concurrent incremental collection



(f) On-the-fly collection



(g) On-the-fly incremental collection

Complications

- We are not responsible for making sure the mutator is thread-safe (and it might not be!)
- In concurrent GC, we must be thread-safe w.r.t. mutator
- In parallel GC, we must avoid duplicating work

Toolbox

- Atomic swap
- Compare-and-swap/test-and-set
- Atomic increment/decrement
- Mutexes, semaphores, etc

Memory consistency

Reordering	Alpha	x86-64	Itanium	POWER	SPAR[illegible]	x86
R → R	Y		Y	Y	[illegible]	
R → W	Y		Y	[illegible]		
W → W	Y		Y	[illegible]		
W → R	Y	Y	[illegible]	Y	Y	Y
Atomic → R	Y	[illegible]	Y	Y		
Atomic → W	Y	[illegible]	Y	Y		
dependent load[illegible]	[illegible]					

It's annoying and I don't want to talk about it.

Tal[illegible] Memory consistency models and possible reorderings. A Y mea[illegible] that the indicated happens-before order is not necessarily enforced.

Parallel GC

- Scanning objects is mostly independent
- Not embarrassingly parallel: Objects may be reachable through multiple paths, so threads must communicate
- Multiple threads scanning an object is at least a waste of time (mark-and-sweep), possibly a catastrophe (copying)

Parallel GC

- Each task has different requirements:
  - Mark phase: Must spread scanning work amongst threads
  - Sweep phase: Relatively easy to simply divide up pools, but can cause starvation
  - Copying phase: Same problems as mark phase, plus parallel allocation
  - Compaction phase: Similar to sweep

Is it worth it?

- Parallelism creedo #1:
  Parallel code is so much slower than sequential, it may take many cores to gain performance
- The best parallel GCs estimate runtime and switch to sequential when no benefit is expected

Synchronization

- Synchronization operations are what make parallelism so slow
- We avoid synchronization whenever possible
- Redundant work is often better than synchronization, so long as the result is correct

Parallel root scanning

- Usually easy: Just divide the roots into equal-sized chunks
- This can create starvation, if some chunks have more references
- Another technique is to divide as stripes: Word 1 goes to thread 1, word 2 → thread 2, … word *n* → thread *n%THREAD_COUNT*

Parallel mark

- Our three mark phases are:
  - Acquire a reference from the worklist
  - Check the object’s mark and mark it
  - Add the object’s references to worklist
- Even with no synchronization, result would be consistent

Parallel mark

```
markPhase(worklist):
  while loc := worklist.pop():
    obj := *loc
    if obj.mark: continue
    obj.mark := 1
    (add obj's references to
     worklist)
    if worklist.empty():
      (synchronize to get more work
       from other threads)
```

Parallel mark

Use a thread-local worklist to avoid synchronization on acquiring work

```
markPhase(worklist):
  while loc := worklist.pop():
    obj := *loc
    if obj.mark: continue
    obj.mark := 1
    (add obj's references to
     worklist)
    if worklist.empty():
      (synchronize to get more work
       from other threads)
```

Parallel mark

```
markPhase(worklist):
   while loc := worklist.pop():
      obj := *loc
      if obj.mark: continue
      obj.mark := 1
      (add obj's references to
       worklist)
      if worklist.empty():
         (synchronize to get more work
          from other threads)
```

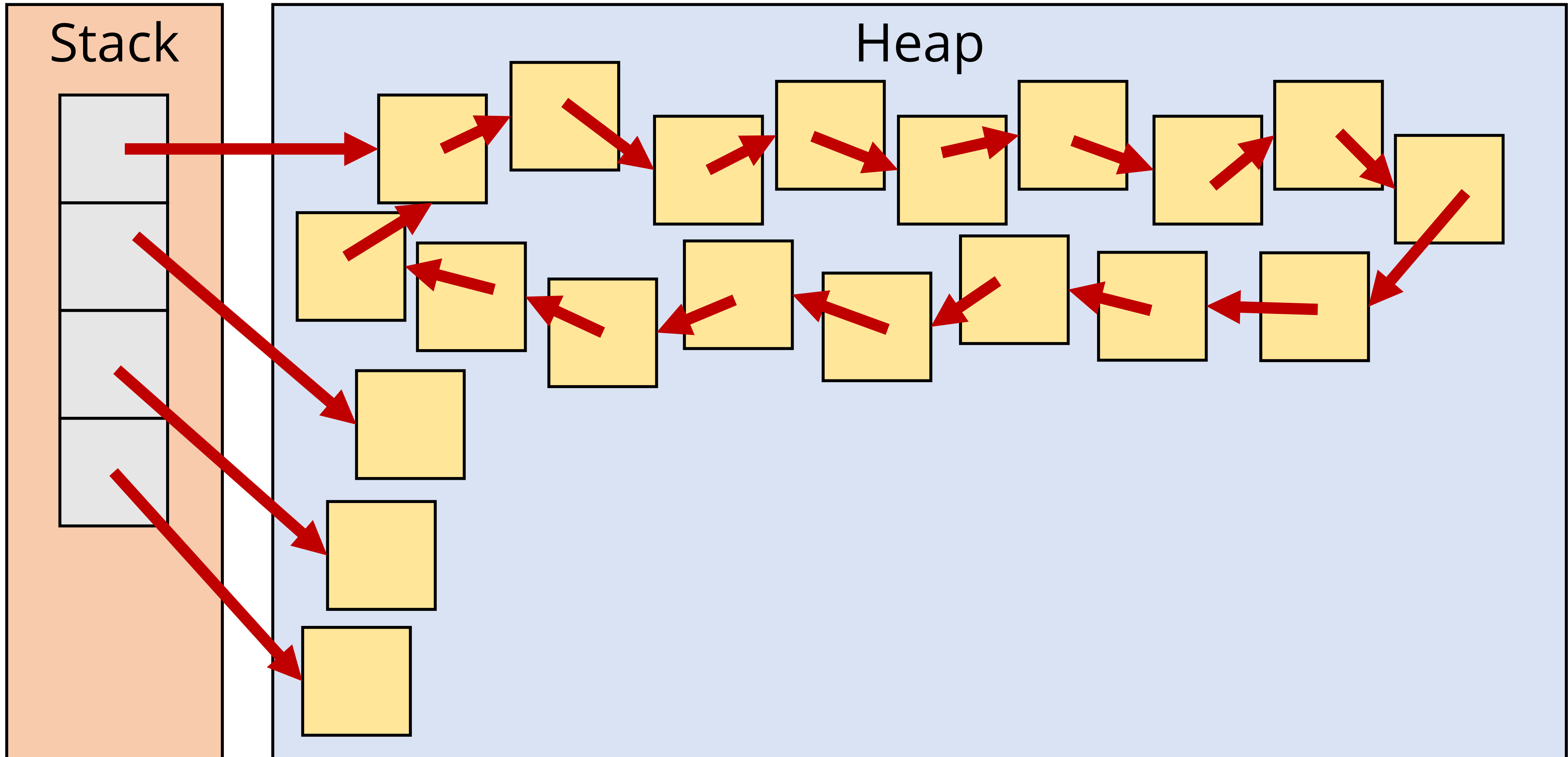
Use a thread-local worklist to avoid synchronization on acquiring work

We could use test-and-set here for guaranteed semantics, but with no synchronization the result is still correct

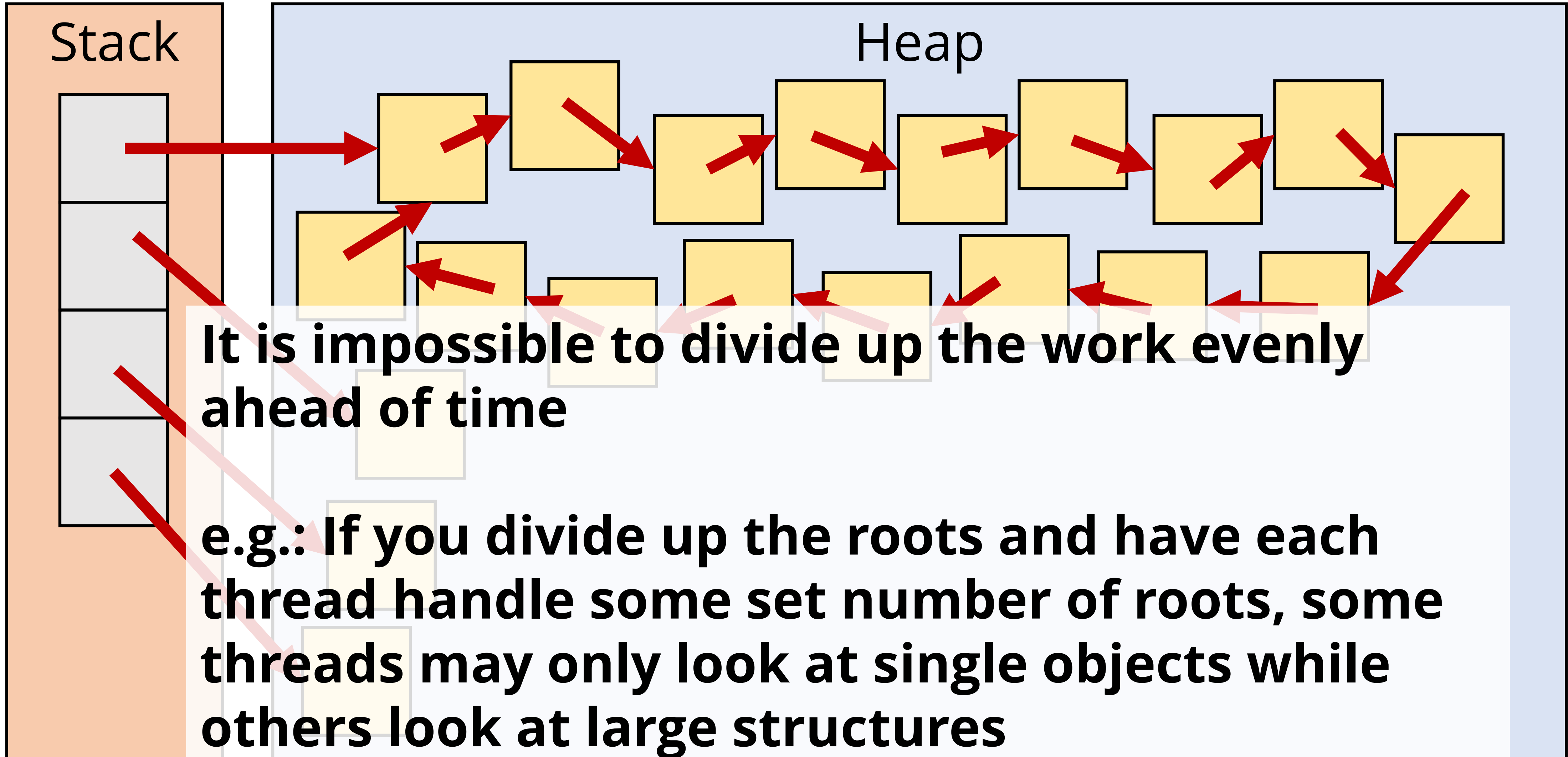
Work synchronization

- With thread-local worklists, `worklist.empty()` does not mean there's no work to do: Another thread may still have more work
- Would be nice: Who cares! Just wait for every thread to be done with their work

Load balancing



Load balancing



It is impossible to divide up the work evenly ahead of time

e.g.: If you divide up the roots and have each thread handle some set number of roots, some threads may only look at single objects while others look at large structures

Work sharing/stealing

- Need a way of either
  - Sharing our excess work with other threads, or
  - Stealing work from threads with an excess
  - Or both
- Thread worklists should be normally unsynchronized, but filled by sharing/stealing

```
shared jobs[N] := (initial work)
shared busy[N] := [true, …]
shared allDone := false
shared signal := new Signal()

worker():
  while true:
    while !jobs[me].empty():
      if (some thread j is not
          busy):
        some := jobs[me].half()
        sendJobs(some, j)
      else:
        jobs[me].pop().do()
    busy[me] := false
    signal.post()
    while jobs[me].empty() and
          !allDone: wait
    if allDone: return
    busy[me] := true
```

```
sendJobs(some, j):
  jobs[j].atomicPush(some)
  while !busy[j] and
        !jobs[j].empty(): wait

detect():
  anyActive := true
  while anyActive:
    signal.wait()
    anyActive := any busy[N]
  allDone := true
```

```
shared jobs[N] := (initial work)
shared busy[N] := [true, …]
shared allDone := false
shared signal := new Signal()

worker():
  while true:
    while !jobs[me].empty():
      jobs[me].pop().do()
    if (some thread j has plenty
        of jobs):
      some := jobs[j].atomicHalf()
      jobs[me].push(some)
    busy[me] := false
    signal.post()
    while (no thread has jobs
           to steal) and
          !allDone: wait
    if allDone: return
    busy[me] := true
```

```
detect():
  anyActive := true
  while anyActive:
    signal.wait()
    anyActive := any busy[N]
  allDone := true
```

Return to parallel mark

- Remember: Needed work stealing for parallel mark
- A direct implementation of work stealing works, but still a lot of synchronization
- We reduce synchronization by dividing up our queue

```
shared stealQueue[N]
threadLocal worklist

acquireWork():
  if !worklist.empty():
    return
  lock(stealQueue[me])
  worklist.push(
    stealQueue[me].popHalf())
  unlock(stealQueue[me])

  if worklist.empty():
    foreach thread j:
      if tryLock(stealQueue[j]):
        worklist.push(
          stealQueue[j].popHalf())
        unlock(stealQueue[j])
        if !worklist.empty():
          return
```

```
markPhase():
  while true:
    while worklist.pop():
      (the usual)
      generateWork()
    acquireWork()

generateWork():
  if stealQueue[me].empty():
    lock(stealQueue[me])
    stealQueue[me].push(
      worklist.popHalf())
    unlock(stealQueue[me])
```

Note: `detect`, `busy` and termination are broadly the same as before, left as exercises for the reader.

Parallel sweep

- One thread sweeps one pool at a time
- Either work-stealing for pools, or accept synchronization
- Build thread-local free-lists to avoid synchronization
- Note: Free-list is now unordered. So much for easy coalescence!

Parallel copying

- Double-copying an object is catastrophic
- Synchronizing on the forwarding pointer seems unavoidable
- We can avoid synchronizing on allocation by distributing pools per thread
  - Or, buffers within pools

Parallel copying

```
copyPhase(worklist):
  topool := (acquire pool for thread)
  while loc := worklist.pop():
    obj := *loc
    if obj.forward: continue
    while topool.freeSpace() < obj.size:
      topool := (acquire pool for thread)
    if !testAndSet(&obj.forward, NULL, topool.allocPtr):
      *loc := obj.forward
      continue
    (copy obj to newObj in topool)
    *loc := newObj
    (add newObj's references to worklist)
    if worklist.empty():
      (synchronize to get more work from other threads)
```

Parallel copying problems

- Contention on forwarding pointer
- Locality guarantees not as nice as traditional copying
- Because threads own pools, we may find tospace full with objects left to copy: Need to give those to other threads

Parallel copying notes

- Thread partitioning helps somewhat: If we have thread partitioning, give each thread the objects in its pools
- With immutable objects[1], we can eagerly copy and race on forwarding

[1] But note, even in a language like Haskell, compiler optimizations sometimes eliminate immutability

Parallel compaction

- Remember our three compaction sweeps:
  - 1: Imaginary compaction to determine forwarding addresses
  - 2: Update references to forwarding addresses
  - 3: Compact

Parallel compaction

- Each phase is easily parallelized *if* we don't perfectly compact, just compact pools
- Again, threads perform per-pool phases
- All threads must complete phase 1 before any start phase 2, and 2→3

Final parallel thoughts

- Tracing is more complicated than sweeping
- Termination conditions are what make balancing so difficult
- It is quite possible to gain zero benefits due to synchronization