Algorithms for Dynamic Memory Management (236780)

Lecture 10

Lecturer: Erez Petrank

Last Week

- Cycle collection
- Compressor

Topics Today

- Allocation techniques
- Parallel GC
  - Parallel Mark-Sweep
  - Parallel Copying

Allocation Techniques

Allocation techniques

- Fragmentation
- Some basic notions and techniques
- Doug Lea's allocator
- Boehm's allocator
- Allocation caches (IBM)
- Immix [Blackburn, Mckinley 08]
- Hoard [Berger, McKinley, Blumofe, Wilson 00]

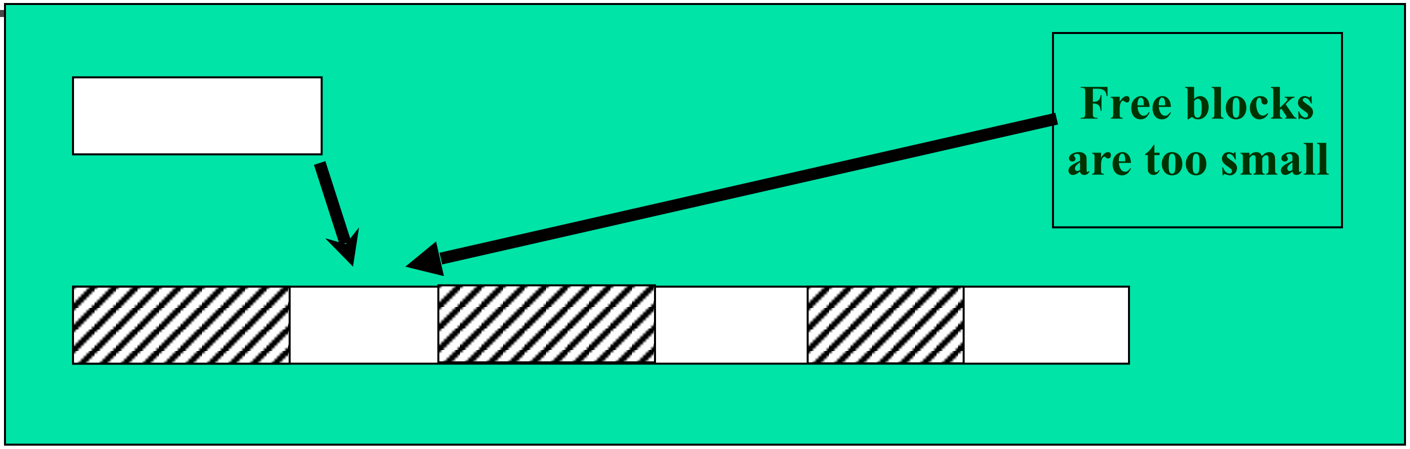
Allocator is measured by

- Speed of allocation
- Fragmentation
- (Speed of reclamation)
- Cache-conscious placement.

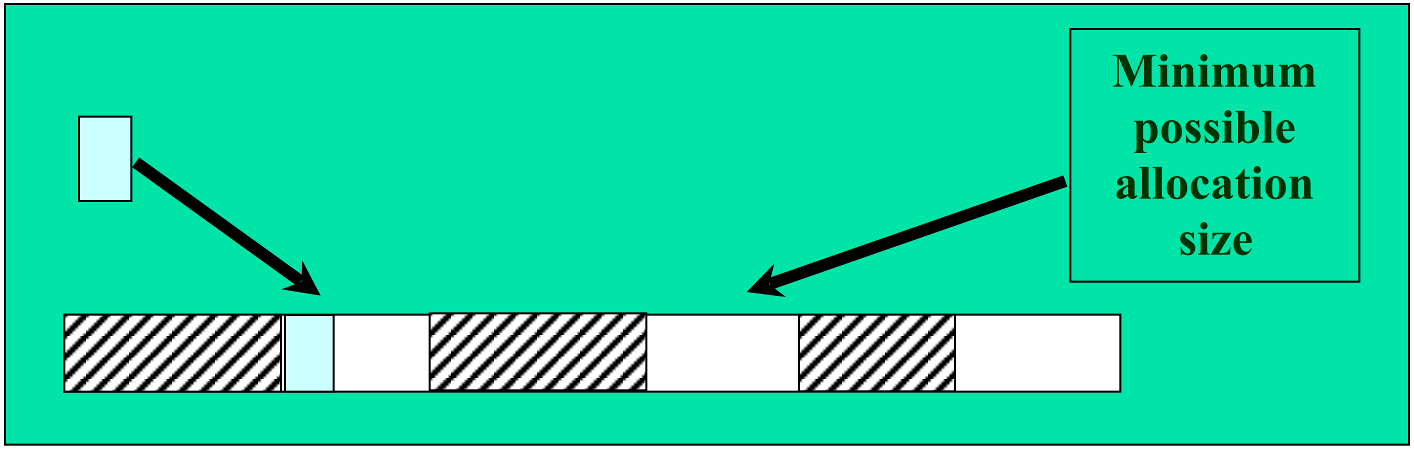
Fragmentation

- Inability to reuse memory that is free
  - Severity determined by distribution of holes, and requests of the program.
  - Caused by reclamation or allocation policies (e.g., allow only certain sizes on one page).
- External fragmentation: holes outside the objects.
- Internal fragmentation: allocated area is larger than requested area.

External frag.



Internal frag.



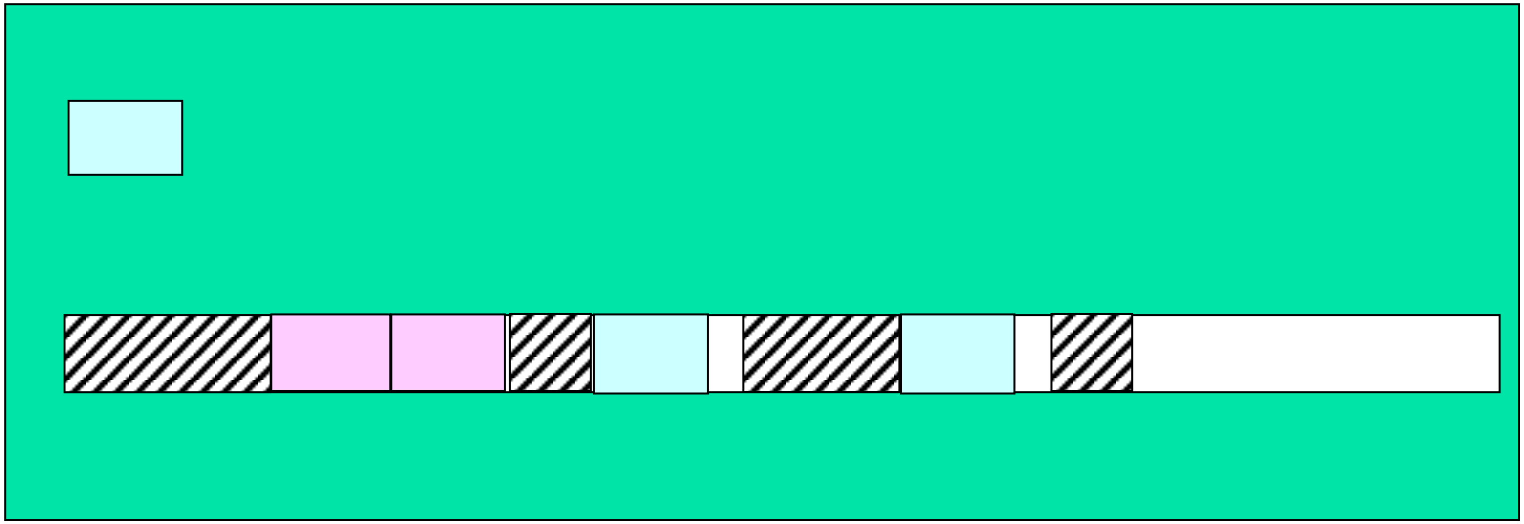
Basic Notions and Techniques

- Various fits: best fit, first fit.
- Indexed fits
- Segregated free lists
- Boundary tags

Best fit

- Allocate in the smallest free block that may satisfy the request.
- In practice,
  - Exploit most of the used “hole”.
  - remainder will be quite small and perhaps unusable.

Best Fit Illustrated



Best Fit (Cont.)

- Naïve implementation: search the whole free-list (linear complexity, unacceptable).
- Common implementations:
  - balanced binary trees, or
  - keep a list of available blocks for each possible allocation size (“indexed fits” or “segregated fits”).
- Note difference between policy and implementation
- Best fit has low fragmentation with typical benchmarks.

First Fit

- Allocate in the first sufficiently large free block.
  - Typically address-ordered,
  - can be “free-list ordered”, or other.
- Search, split found block, put remainder on free-list.

First Fit behavior

- Lots of small blocks near the beginning of the list.
- These “splinters” (שבבים, רסיסים) increase fragmentation and may increase search times.
- But: normally maintains very low fragmentation.

Indexed Fits

- Use an indexing data structure to obtain efficient searching of a desired policy.
- Examples:
  - Best fit with a balanced tree.
  - Buddy systems
  - Bitmap fits – use a bitmap to search for free space…

Modern Allocators

Using Header Fields

- Most allocators use a hidden “header” field within each allocated area to store useful information like:
  - Size of the block, whether the block is allocated, its class object, locking info, hash info, garbage collection info (reference count, mark-bit) etc.

Coalescing via Boundary Tags

- When object is freed, try to coalesce its free space with adjacent free spaces.
- For coalescing efficiency, allocated areas may also contain a “footer” field with size of block, denoted *boundary tag*.
- When a block is freed, examine footer of preceding block and header of following block for coalescing.
- Space saver: use footer only if object is free. Use a bit in the next object header to indicate if it is.

Segregated Free Lists

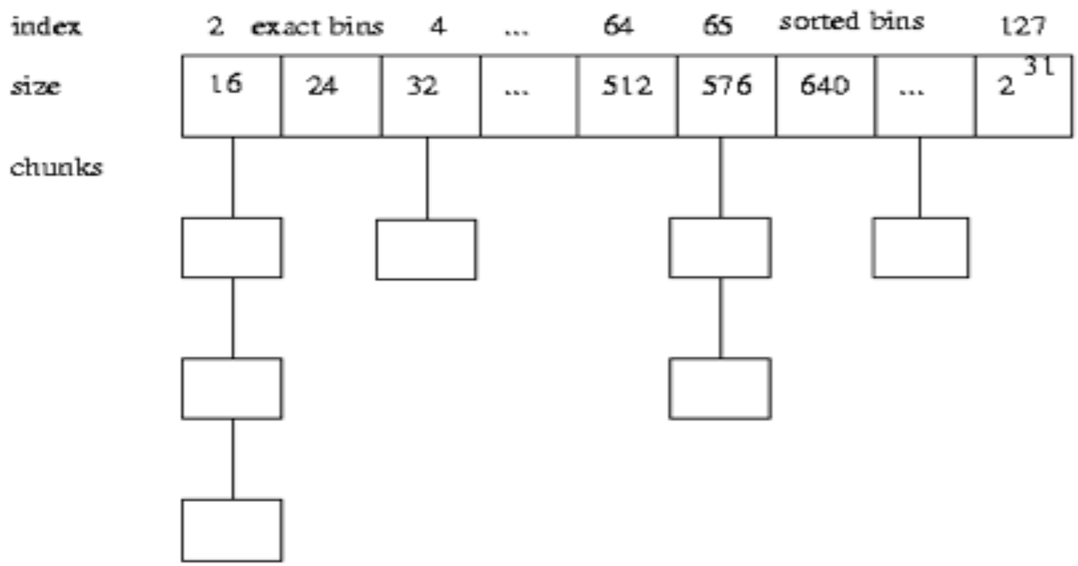
- Typical structure: an array of free lists.
  - Each list holds free chunks of a particular size.
  - A freed chunk is pushed to the appropriate free list by the collector.
  - Allocation uses appropriate list.
  - A pool of free blocks is kept aside.
- Preferred implementation for approximate best fit.

Segregated Free Lists Variations

- Each free list has a range of sizes for allocation.
  - Search for a chunk in the list using best fit, first fit, or next fit. (Typically --- first fit.)
- Number of lists.
  - A small number of lists may increase internal fragmentation or allocation time.
  - A large number of lists may increase space overhead.

Algorithm used

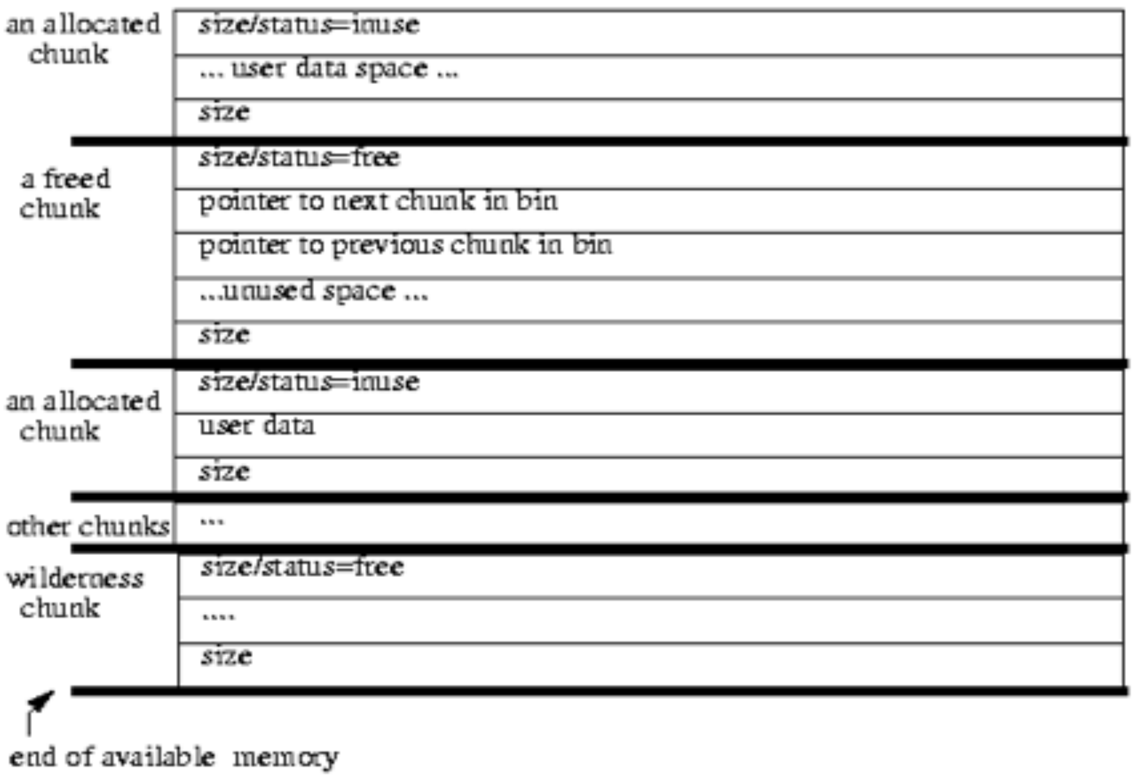
- Segregated free lists - Available chunks are maintained in bins, grouped by size.



- (Almost) Best fit

Doug Lea's Malloc

- Boundary Tags for coalescing & traversing starting from any chunk in any direction.



Improving Locality

- Use the following modification:
  - If cannot find space in the desired bin, attempt to allocate from the space most recently used for split.
  - If that fails, switch to best fit (and record the new block found for future allocations).
- Resulting algorithm is almost best fit.
- Wilderness Preservation:
  - The ”wilderness” is the free space at the end of the heap.
  - This chunk will be used only if no other smaller suitable chunk exists.
- http://gee.cs.oswego.edu/dl/html/malloc.html

Boehm's allocator (and collector)

- http://www.hpl.hp.com/personal/Hans_Boehm/gc/
- Collector by Boehm-Demers-Wiser.
- May be used as a leak-detector
- Distributed with the GNU compiler
- Available for most standard PC and UNIX platforms, Win32, OS/2, and UNIX environments.
- The collector uses a mark-sweep algorithm.
  - With incremental and generational support.

Block-Oriented Segregated Free Lists

- Use segregated free lists, but with blocks.
- Heap is partitioned to blocks (typically 4KB = page size).
- A block has only objects of same size (e.g., 24 bytes).
- When a list of free chunks is empty, a new block is obtained and split into same size objects.

Segregated Free Lists, Block Oriented

- Advantages
  - Shortened headers: (no need for size).
  - Efficient in time and space since typical programs use few sizes.
  - "Better order" in the sense that small objects are not scattered all around. This may reduce fragmentation.
- Disadvantages:
  - But, making a page take only one size, may increase fragmentation.

Block Maintenance

- Free blocks are maintained in a tree sorted by address.
- Two level allocator:
  - Large objects are allocated from tree of blocks, first-fit.
  - Small objects are allocated inside the blocks.
- Sweep returns
  - an empty block to the tree of blocks
  - empty chunks inside a non-empty block to the appropriate list.

Parallel Allocation

Allocation for SMP's

- Boehm's allocator and segregated free lists in general can be extended for a multiprocessor.
  - Typically, by maintaining segregated free lists per thread.
- Next:
  - Allocation caches (IBM).
  - Immix.
  - HOARD.

Allocation caches

- Two goals:
  - Reduce contention on allocation.
  - Allow “bump-pointer” allocation for mark-sweep.
- Method: let each thread obtain a “local cache” using synchronization.
- After obtaining the local cache: allocate (small objects) from it locally with a bump pointer.

Method with Mark-Sweep

- All available spaces are kept in a free-list, created by sweep.
- When a local cache is needed, the first large-enough space is taken via first-fit.
- There is a minimum and maximum size for a cache.
  - Minimum – because we do not want to switch caches too often. Switching involves synchronization.
  - Maximum – because we do not want one thread to use all free space as its own cache, starving the other threads.
- All small objects are allocated from the cache.

The Free List

- If a free chunk on the list is too large, only a piece of it is taken for the cache, and the rest is left in the free list.
- Allocation of larger objects (say, more than half the minimum cache size) is done directly from the free-list via first-fit.
- A simple optimization: sweep does not put small spaces in the free list. All free chunks are large enough to serve as caches.

Allocation Caches Properties

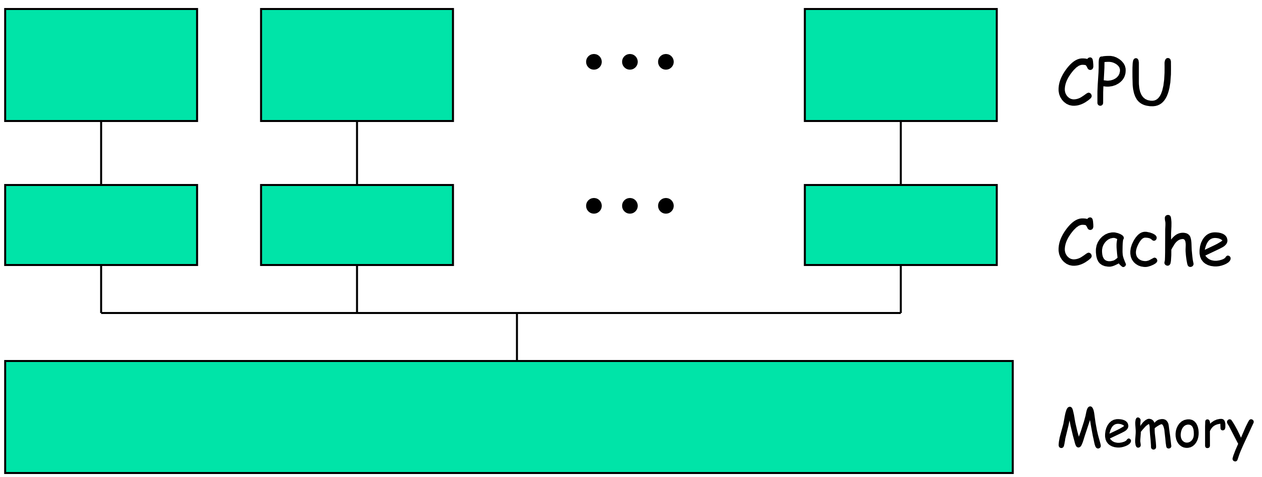
- Cache behavior: it is believed that allocating sequentially is very good for program locality.
- Local caches provide bump-pointer allocation to mark-sweep.
- It is fast and cache-friendly.
- Most allocations are executed locally in the local cache.
- Synchronization is seldom and thus contention is low.

Hoard

- A Scalable Memory Allocator for Multithreaded Applications [Berger- McKinley-Blumofe-Wilson 01].
- Download from http://www.hoard.org .

- Goal: achieve efficiency and scalability on a multiprocessor.
- Strategy: avoid contention, avoid false sharing.

An SMP



Costly access to memory is ameliorated by the use of fast caches.

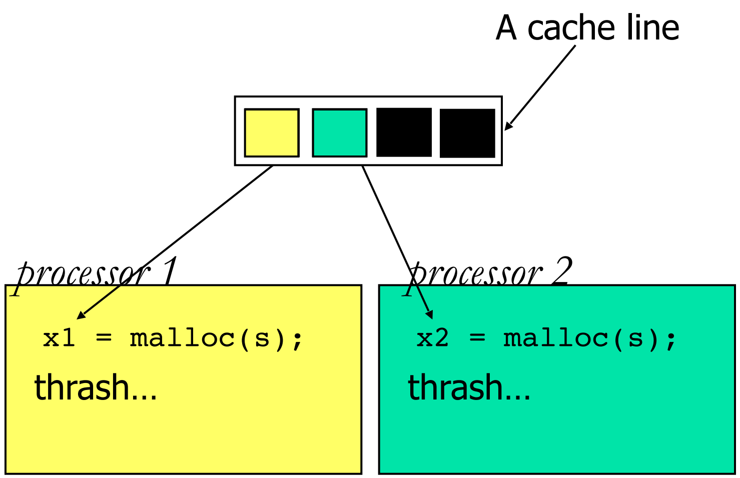
What is Sharing?

- If one object is used by several threads, then its content must be repeatedly consolidated between the caches.
- This is done by the cache coherence protocol.
- Use of caches is not as efficient in this case.

What is False Sharing?

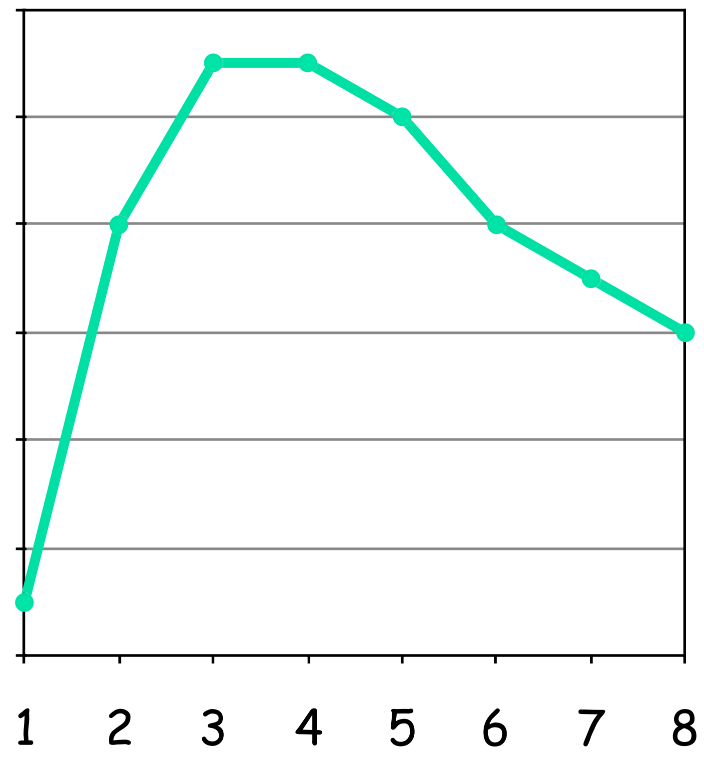
- Suppose a cache line holds two objects O1, O2 (or more) allocated by two threads T1, T2 (or more).
- When T1 and T2 access their objects, sharing is falsely created because these objects which reside on the same cache line.
- “Thrashing”.

Allocator-Induced False Sharing



Implication

- No scalability

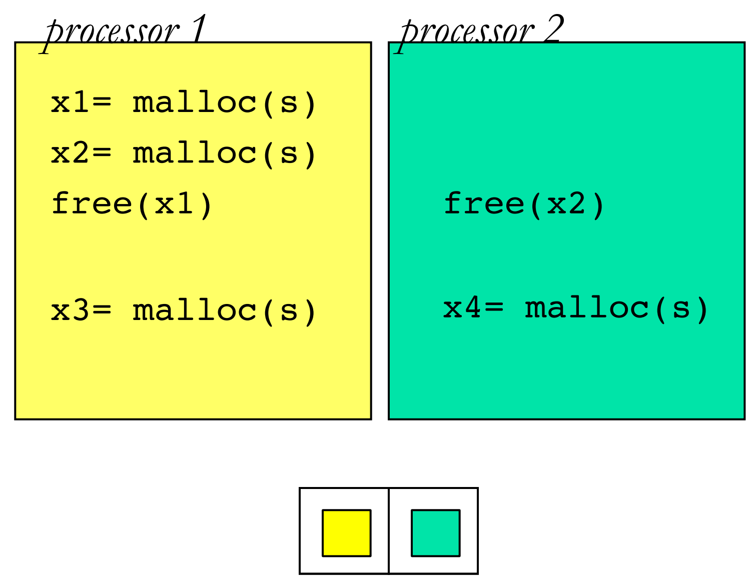


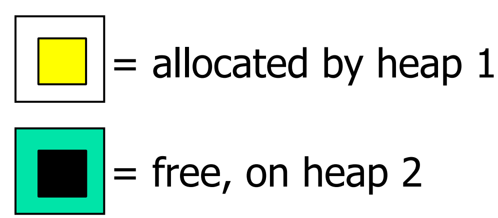
Using Local Heaps (Only)

Using one heap per thread:

- malloc gets memory from the processor's heap or the system
- free puts memory on the processor's heap

Creates thrashing.





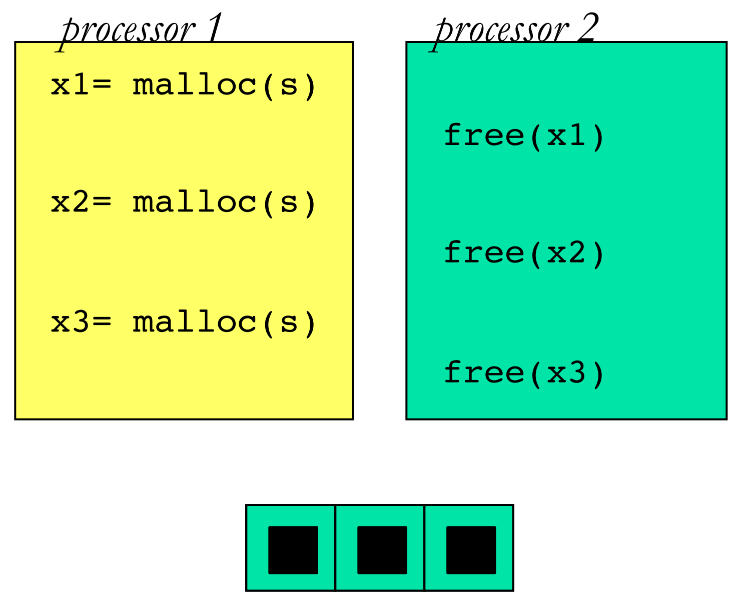
Problems with Pure Local Heaps

Moreover, memory consumption can grow without bound!

Producer-consumer:

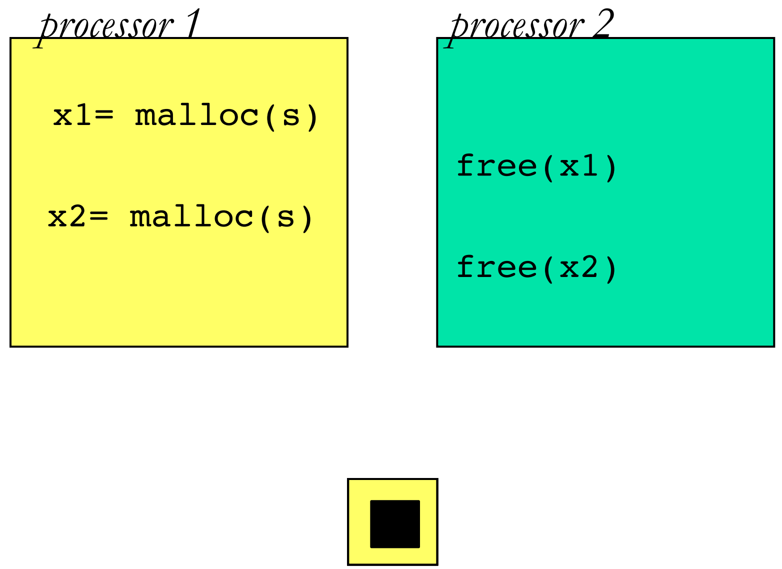
processor 1 allocates

processor 2 frees



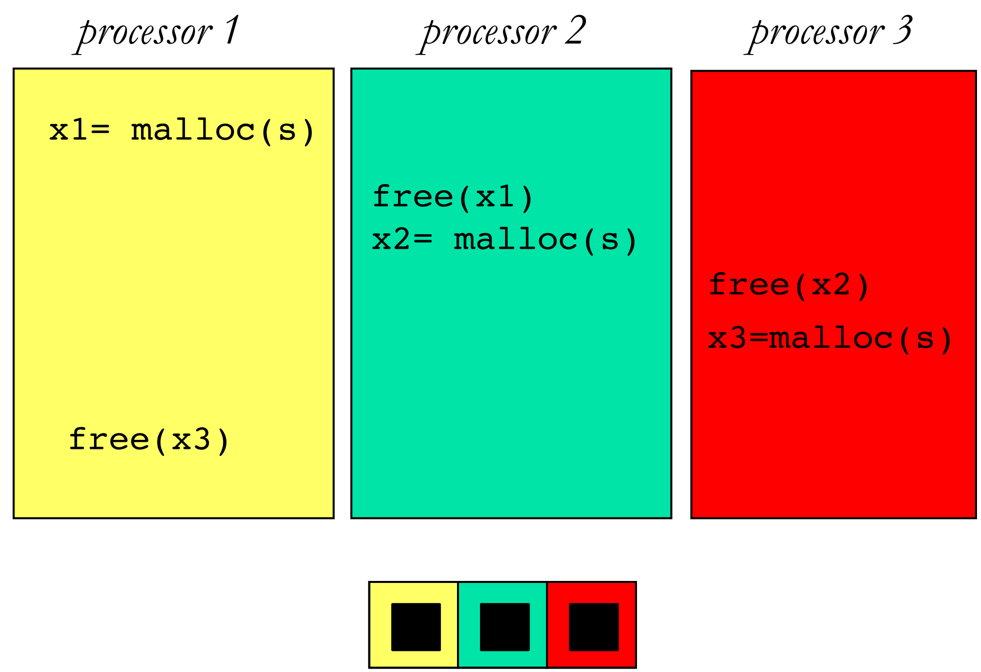
Allowing Ownership

- free puts memory back on the originating processor's heap.
- Avoids unbounded memory consumption



Still, Problems

- memory consumption can blow up by a factor of P.
- Problem: free chunks "belong".
- Round-robin producer-consumer:
  - processor i allocates
  - processor i+1 frees.



Main Ideas in Hoard

- Use thread local heaps consisting of page-sized blocks.
  - Avoid contention on allocations.
  - Avoid false sharing: cache lines do not split between blocks.
- Each local heap employs block-oriented segregated free lists.
  - Namely, it consists of blocks, each holding objects of one size.

Main Ideas in Hoard

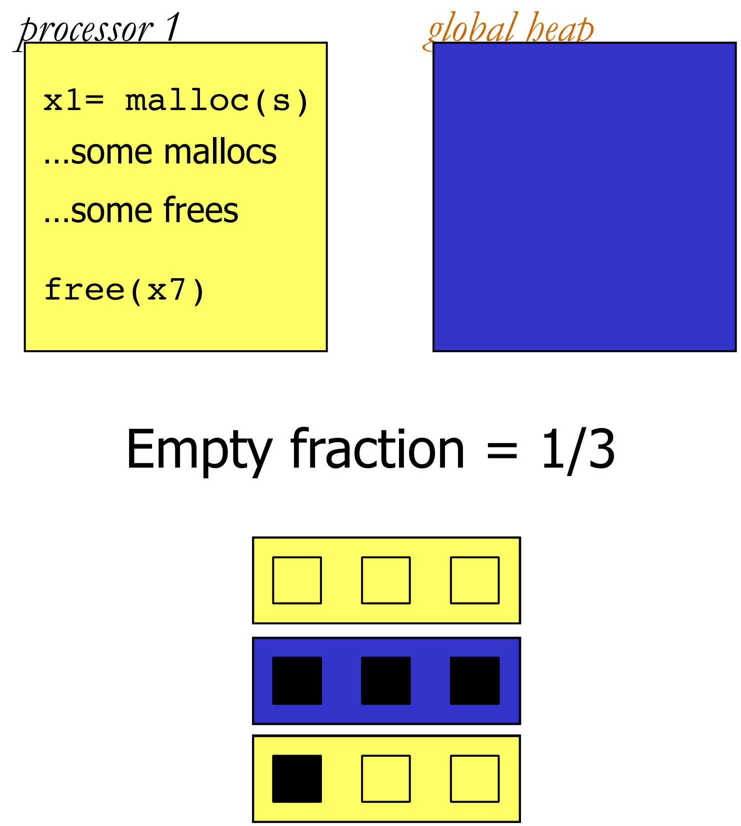
- Freed blocks stay in local heaps (available for allocation)
- When the fraction of free memory in the local heap exceeds the empty fraction, blocks are returned to a global pool of blocks.
  - Avoid blowup in memory consumption, allow reuse of memory.
- When local heap is full, a new block is obtained from the global pool.

Main Ideas in Hoard

- Large objects (more than half a block) are allocated directly from the operating system.
- Small objects are allocated from the thread's local heap in one of its blocks.
- Recycling a block: when a block is completely free the size of its objects may change.
  - Reduce fragmentation.

Hoard Example

- malloc gets memory from a block on its heap.
- free returns memory to its block. If the heap is "too empty", it moves a block to the global heap.

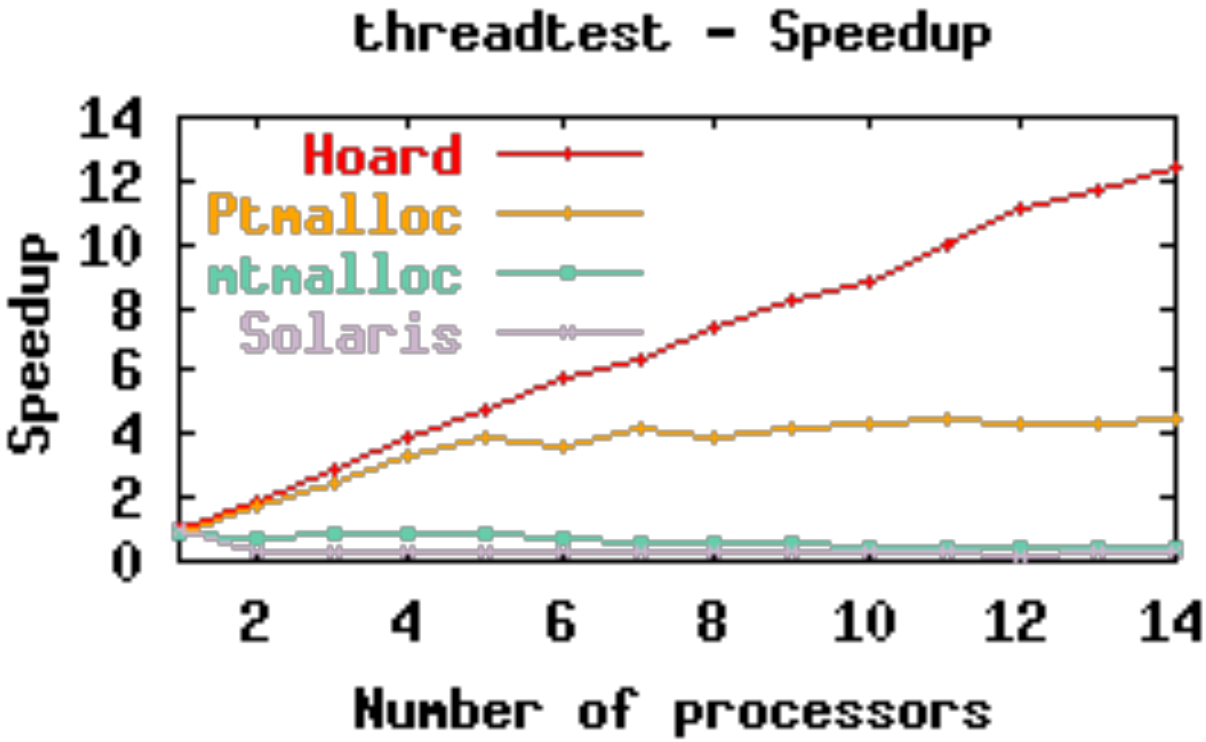


Compared Allocators

- Solaris: Default allocator for Solaris 7
- Ptmalloc: Linux allocator included in the GNU C library.
- MTalloc: a multiple heap allocator included with Solaris 7 for use with multithreaded parallel application.

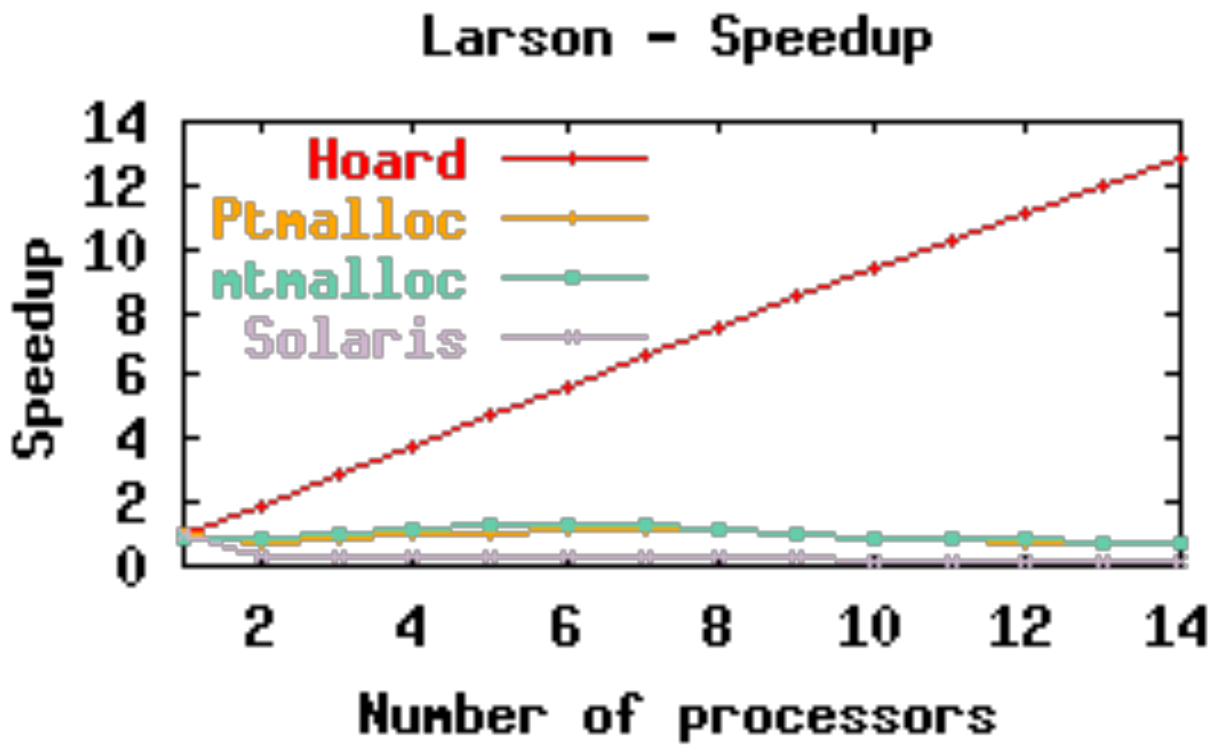
speedup(x,P) = *runtime*(Solaris allocator, one processor) / *runtime*(x on P processors)

Performance: *threadtest*



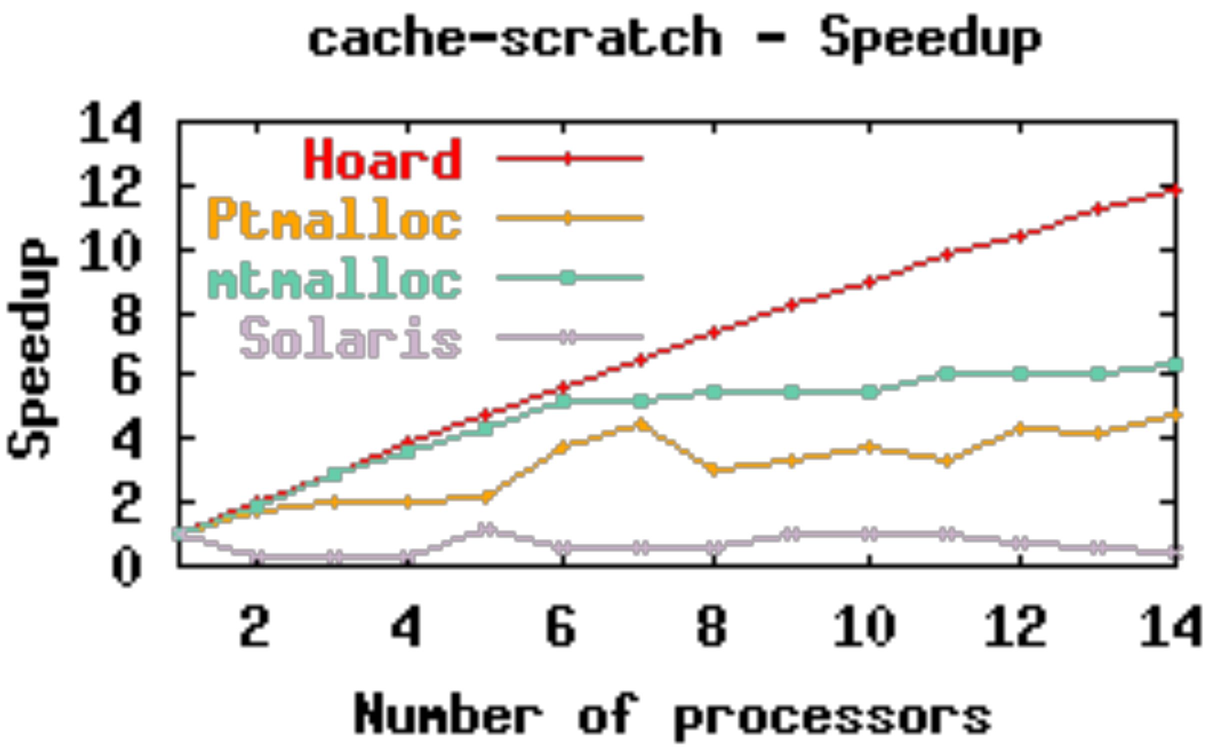
speedup(x,P) = *runtime*(Solaris allocator, one processor) / *runtime*(x on P processors)

Performance: *Larson*



Server-style benchmark with sharing

Performance: *false sharing*



Each thread reads & writes heap data

Fragmentation Results

On most standard uniprocessor benchmarks, Hoard's fragmentation was low:

p2c (Pascal-to-C):	1.20	espresso:	1.47
LRUsim:	1.05	Ghostscript:	1.15

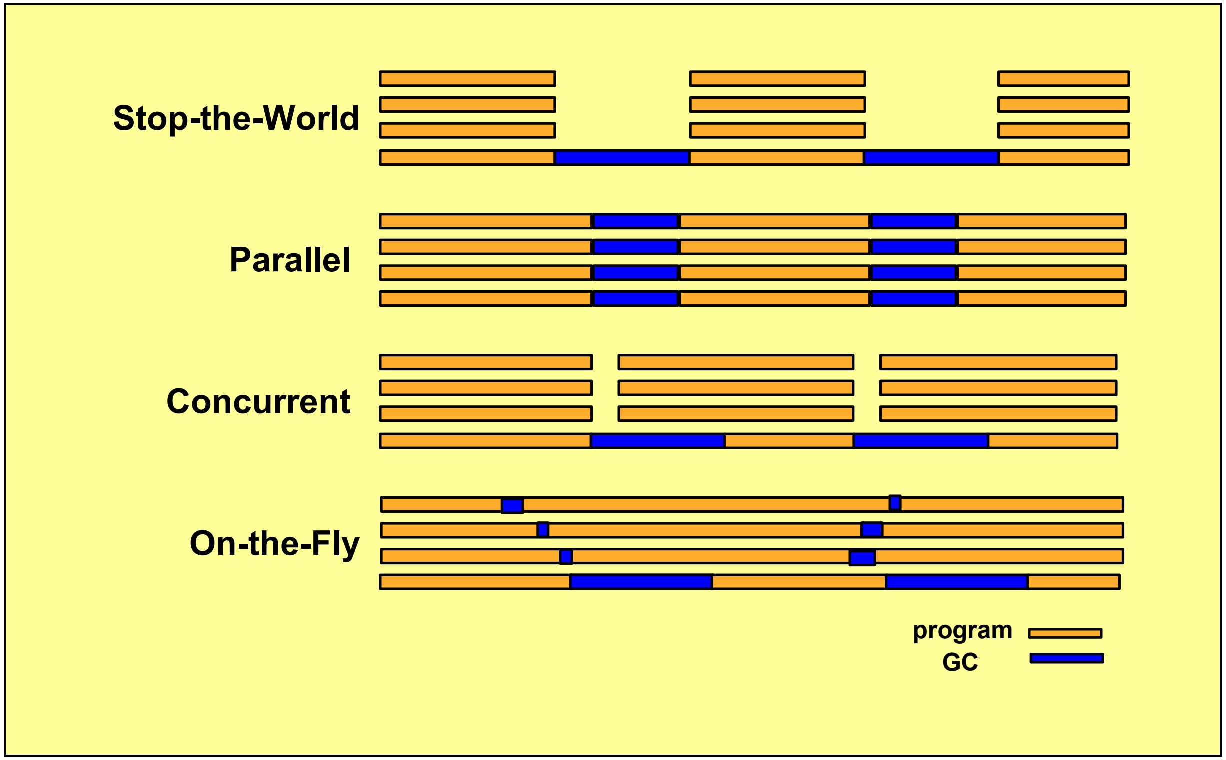
Within 20% of Lea's allocator

On the multiprocessor benchmarks and other codes:

Fragmentation was between 1.02 and 1.24 for all but one anomalous benchmark (shbench: 3.17).

Parallel Garbage Collection

Recall Terminology



Part I: Parallel Mark & Sweep

Motivation

- Concurrent GC may not be scalable enough
  - One collector serves many allocating mutators.
- Parallel collectors do not compete with program, but need to cooperate work on shared resources.
- [Endo-Taura-Yonezawa 1997] “A Scalable Mark-Sweep Garbage Collector on Large-Scale Shared-Memory Machines”
  - First work to study mark-sweep scalability.
  - A 64-way SMP is used.
  - Speedup reported.

What do we want?

- Goals:
  - Scalability
  - Load balancing
  - Locality of reference
  - Simplicity
- Main problem:
  - Work cannot be partitioned statically, must allow collector to dynamically balance work.
  - Standard idea: overpartition.

Base Collector: Boehm GC

- Boehm GC: a C/C++ library for mark-sweep conservative GC.
- Call GCalloc instead of Malloc.
  - GCalloc allocates a large space to allocate in.
  - When space runs out it runs garbage collection.
- Download: http://www.hpl.hp.com/personal/Hans_Boehm/gc .

Base Collector: Boehm GC

- Allocations (via gcalloc)
  - Allocations are serialized.
  - Heap divided into 4KB blocks which are initially in a free list.
  - Large objects are allocated from a free list sorted by address.
  - Small objects are allocated from within blocks.
  - Each block contains objects of the same size

Base Collector: Boehm GC

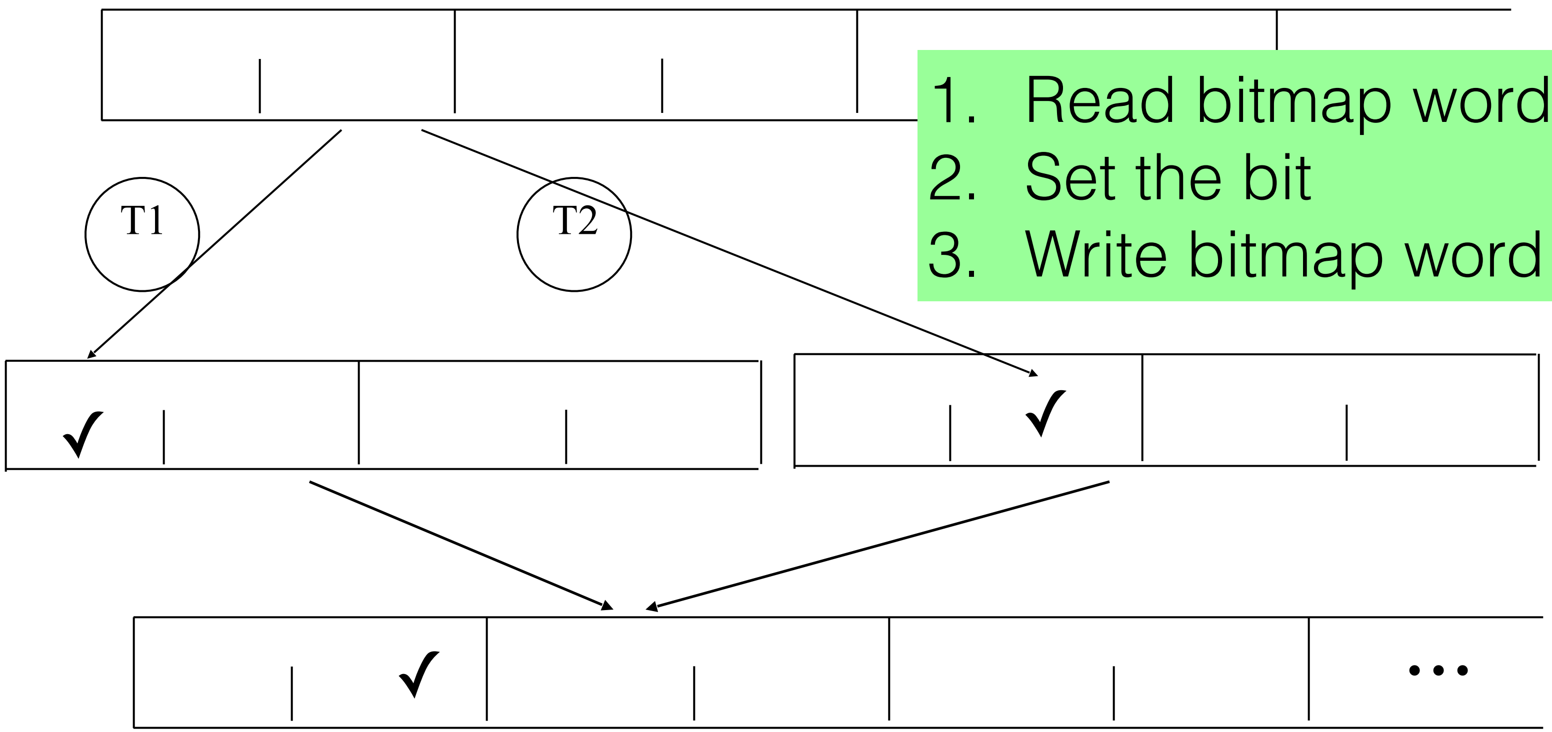
- Every block has a separate mark bitmap (ratio 1:32).
  - Marking requires sync.
- One global markstack requiring sync as well.
- DFS style marking from roots.
- Sweep:
  - If a block is fully empty, then it is returned to free list of blocks. Adjacent blocks are coalesced,
  - If mark bit is set on a heap block, block is kept for future allocations.

Parallel Collector

- Each collector runs on a separate process with a local markstack.
- When GC invoked, all program processes are stopped via a signal.
- Each process marks its local roots and then proceeds marking via its local markstack.
- Marking requires synchronization.

Why Locking of Bitmap is Required

Two threads mark two objects concurrently.
Both objects are mapped to the same bitmap word.



1. Read bitmap word
2. Set the bit
3. Write bitmap word

Parallel Sweeping

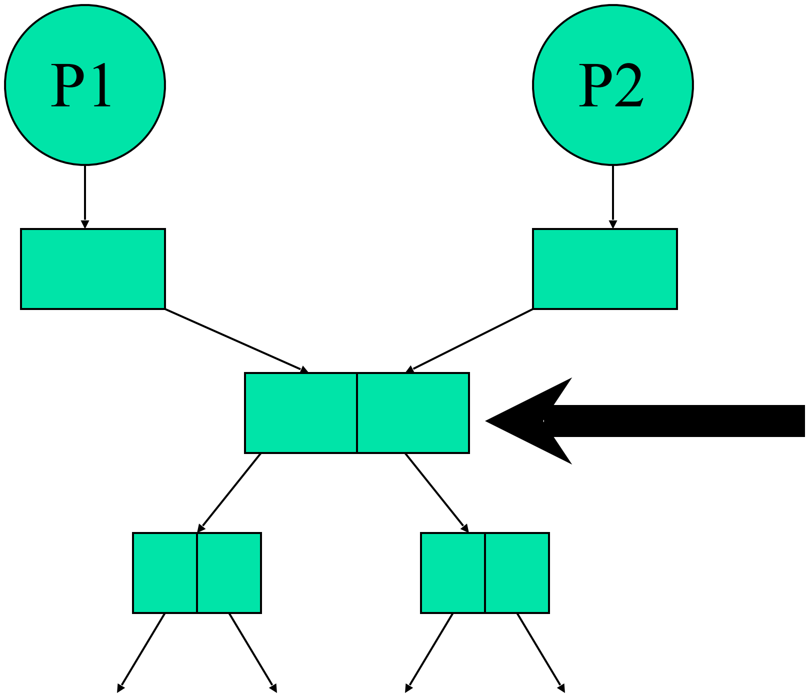
- Each process assigned part of the heap (64 blocks at a time).
  - All blocks that have not become empty are put in local free lists (cheap)
  - All free blocks are first processed locally (sorting, coalescing).
    Then, lock is taken and they are put in global free list.

Naïve Implementation

- This naïve implementation results in hardly any speedup.
- Thus, improvements required.

Introducing Load Balancing

- Problem: Consider the following shared tree:



The process that <u>first</u> marks the tree's root will have to <u>mark the rest</u> of the tree!

Stealable Mark Queues

- Each processor keeps a "stealable mark queue".
- Once in a while, if stealable mark queue empty, move ½ of the jobs from markstack to stealable mark queue.
- If processor idle – search for jobs in stealable mark queues. (Steal ½ of a s.m.q.)
  - ✓ Idle processors help the busy ones
  - ✗ Sync on stealable mark queues
  - ✗ Tougher termination detection

Termination

- Keep a global counter with the number of empty stacks and empty queues.
- Counter is updated whenever a processor fills its mark queue, becomes idle, or obtains a task.
- When counter reaches twice the number of processors – GC ends.

Empirical Evaluation

- With stealable mark queues, the algorithm exhibits at most 12x speed-up on a 64-way SMP.
- Next, 4 improvements.

1: Split Large Objects

- A process that marks a large object (e.g. 400KB) is tied up for a long time (load imbalance)
- Solution: Break objects into 512-byte chunks before inserting into mark stack

2: Skip Locked Queues

- Sometimes many processes attempt to steal from the same queue and must wait to enter the critical section (contention)
- Solution: If lock can't be acquired on first try, give up and go to the next queue.

3: Markbits Test

- Sync. (lock) is used to mark objects
- However, many times the object is already marked !
- Improvement: Read the bit first without sync. If not set, use sync to set it.

4: Improve Termination Detect

- Global counter was used to keep track of empty mark and steal queues (contention)
- Improvement: "Mark Stack Empty" and "Steal Stack Empty" flags kept on each processor. Terminate if all flags are set and global "interrupted" flag is clear.

Empirical Evaluation

- Desired: run various number of processors on the same snapshot of the heap.
- Problem: snapshot depends on number of processors. Not clear how to continue with varying number of processors from the same snapshot.
- Approach: Devise a formula for workload and compare workload/time ratios.

Workload and Speed-up

- Workload of a collection is
  $W = a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4 + a_5x_5$
  x1= no. marked objects marked and scanned
  x2=no. visits to already marked obj's
  x3=no. visits to objects with no pointers
  x4=no. empty heap blocks
  x5=no. non-empty blocks
- Weights were determined by experiment
  - a1=0.50, a2=0.16, a3=0.02, a4=2.0, a5=1.3
- Speed=W/t, (for t = GC elapsed time)
- Speed-up on N proc's = $(W_n/t_n)/(W_1/t_1)$

Benchmark Apps

- BH -- simulates motion of N moving bodies (here N = 5000)
- CKY -- context-free grammar parser (run on 67 sentences, 10-40 words per sentence).

Mark Speed-up Results

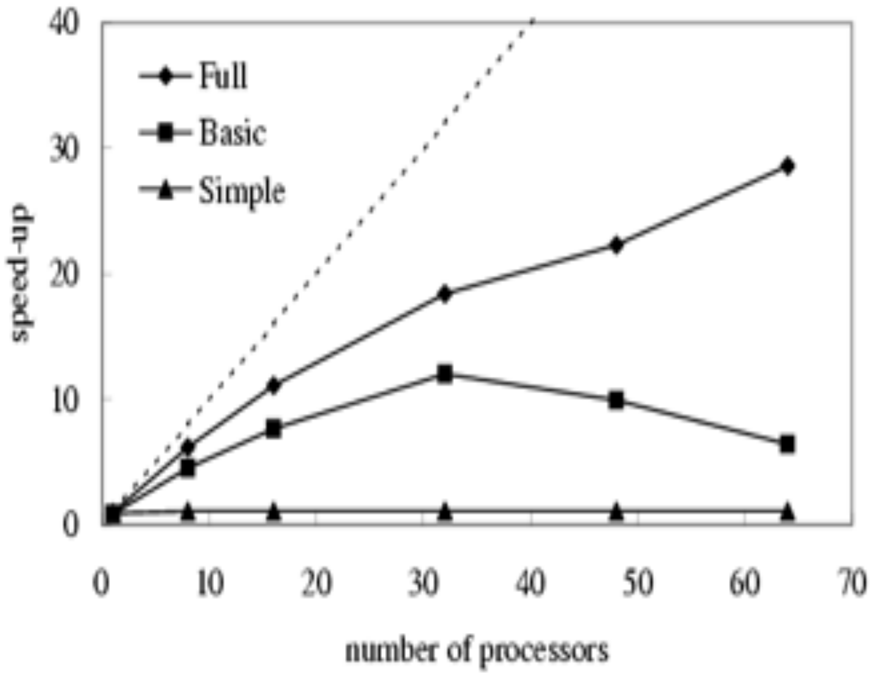


Figure 5: Average marking speed-up in CKY.

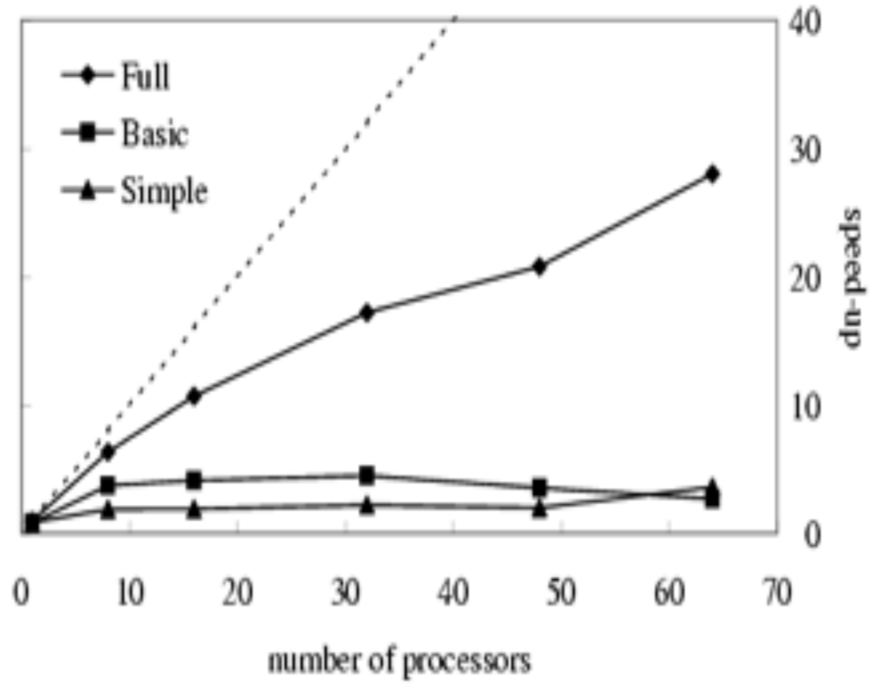


Figure 6: Average marking speed-up in BH.

Indiv. Improvement Effect

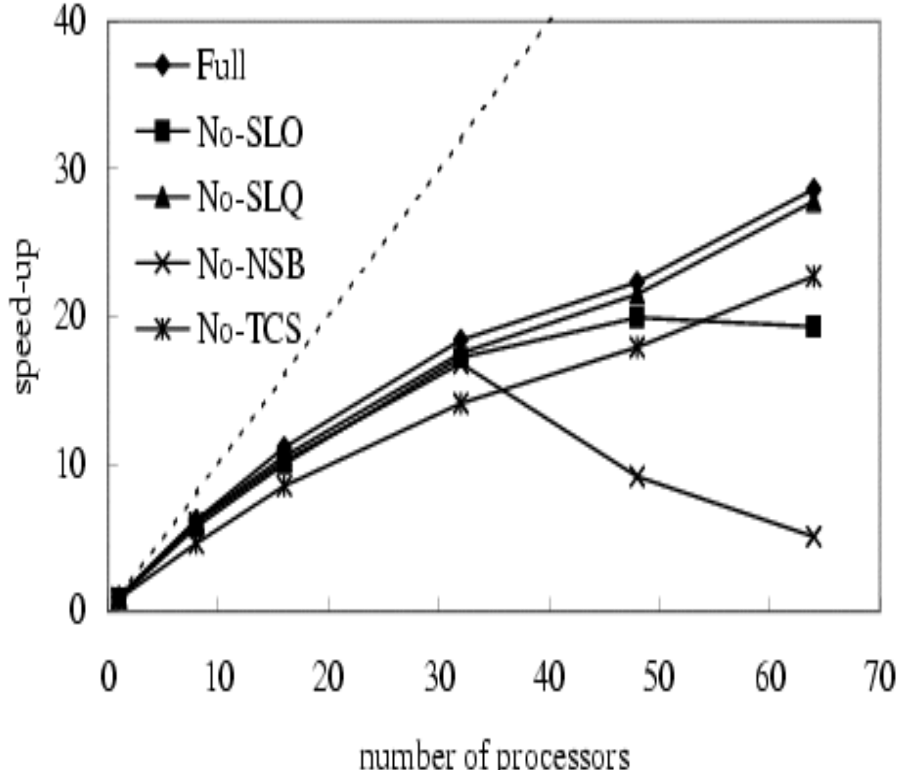


Figure 7: Effect of each optimization in CKY.

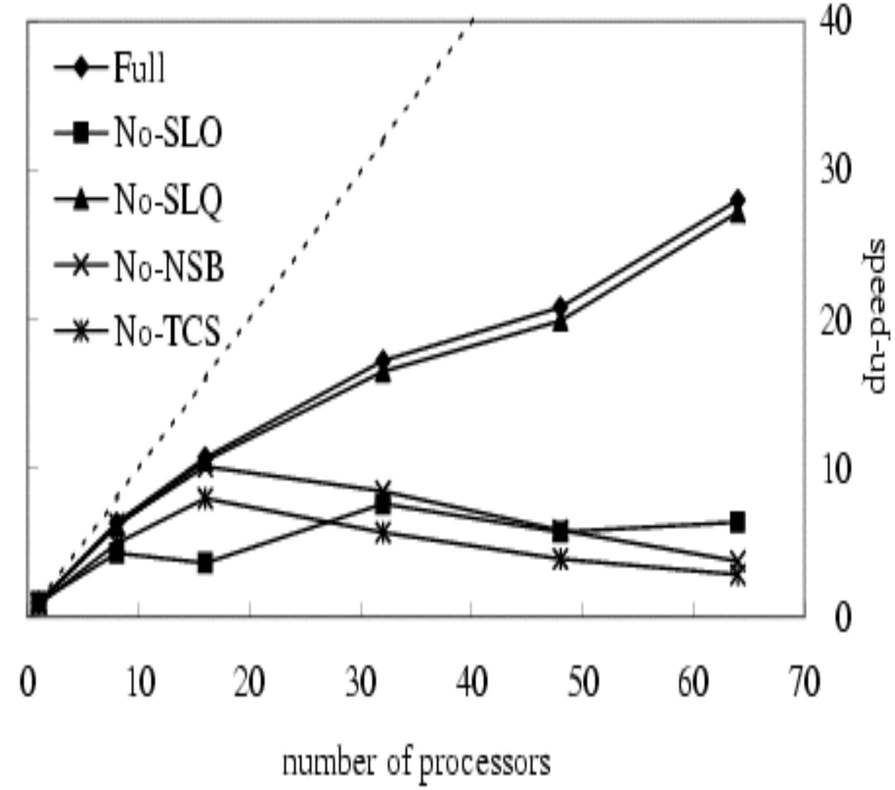


Figure 8: Effect of each optimization in BH.

No-SLO = w/o splitting large objects, No-SLQ = w/o non-blocking mark queues search, No-NSB = w/o improved termination detection, No-TCS = w/o non-blocking bitmap reads

Summary

- Parallelizing a mark-and-sweep collector is possible.
- With the ideas raised in the paper the authors got speed-up around 30 with 64 processors. (Very good!)

Covered So Far

- Mark-sweep basics
- Compaction algorithms
  - Basics (2-finger, Lisp-2, Threaded), parallel (SUN + IBM + Compressor), Concurrent (Compressor)
- Copying: basics + Baker
- Generations and Train
- On-the-fly: Dijkstra and DLG
- Mostly concurrent (IBM)
- Snapshot and sliding-views mark-sweep
  - = concurrent and on-the-fly
- Reference counting (basics + update coalescing + concurrent)
  - Cycle collection
- Allocation techniques
- Parallel collection

Last Class

- Cache Conscious
- Real Time