Allocating memory in a lock-free manner

Anders Gidenstam, Marina Papatriantafilou
and Philippas Tsigas

Distributed Computing and Systems group,
Department of Computer Science and Engineering,
Chalmers University of Technology

Outline

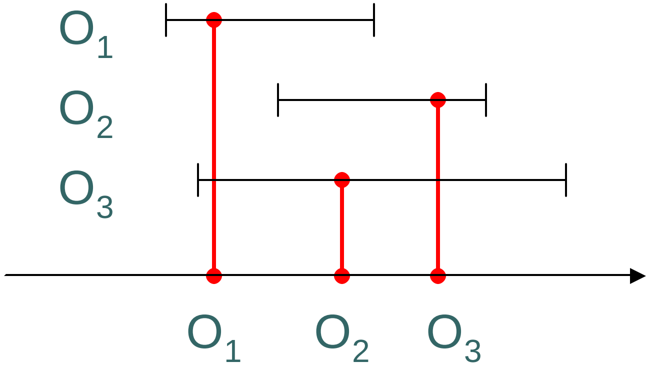
- Introduction
  - Lock-free synchronization
  - Memory allocators
- NBmalloc
  - Architecture
  - Data structures
- Experiments
- Conclusions

Synchronization on a shared object

- Lock-free and wait-free synchronization
  - Concurrent operations without enforcing mutual exclusion
  - Avoids:
    - blocking and priority inversion
  - *Lock-free*
    - At least one operation always makes progress
  - *Wait-free*
    - All operations finish in a bounded number of their own steps
- Synchronization primitives
  - Built into CPU and memory system
    - Atomic read-modify-write (i.e. a critical section of one instruction)
  - Examples
    - Test-and-set, Compare-and-Swap, Load-Linked / Store-Conditional

Synchronization on a shared object

- Desired semantics of a shared data object
  - Linearizability [Herlihy & Wing, 1990]
    - For each operation invocation there must be one single time instant during its duration where the operation appears to take effect.

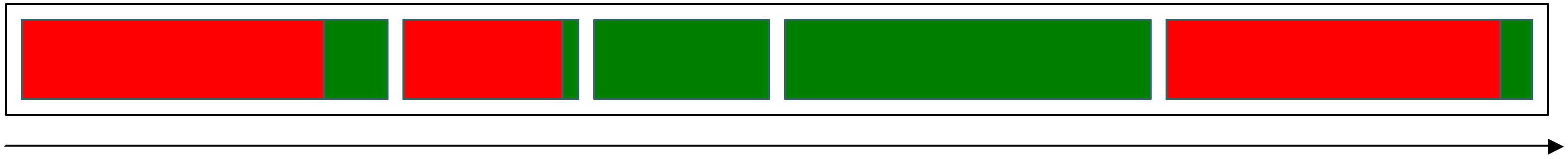


Memory management and lock-free synchronization

- Concurrent memory management
  - Concurrent applications
    - Memory is a shared resource
    - Concurrent memory requests
    - Potential problems: contention, blocking, etc
  - Why lock-free?
    - Scalability/fault-tolerance potential
    - Prevents a delayed thread from blocking other threads
      - Scheduler decisions
      - Page faults etc
    - Many non-blocking algorithms uses dynamic memory allocation
      - => non-blocking memory allocator needed

Memory Allocators

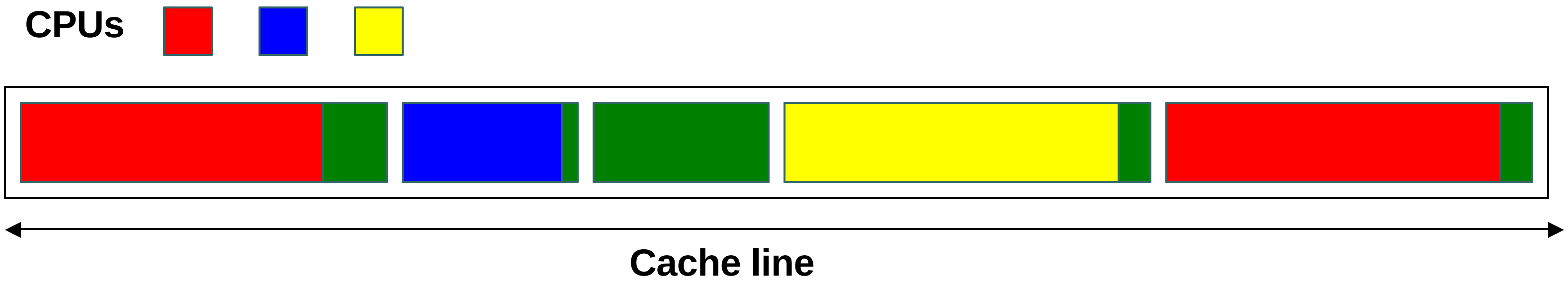
- Provide dynamic memory to the application
  - Allocate / Deallocate interface
- Maintains a pool of memory (a.k.a. heap)
- Online problem – requests are handled in order
- Performance
  - Fragmentation
  - Runtime overhead



Memory address

Concurrent Memory Allocators

- Goals
  - **Scalability**
  - Avoiding
    - **False-sharing**
      - Threads use data in the same cache-line
    - **Heap blowup**
      - Memory freed on one CPU is not made available to the others
    - **Fragmentation**
    - **Runtime overhead**



CPUs 🟥 🟦 🟨

Cache line

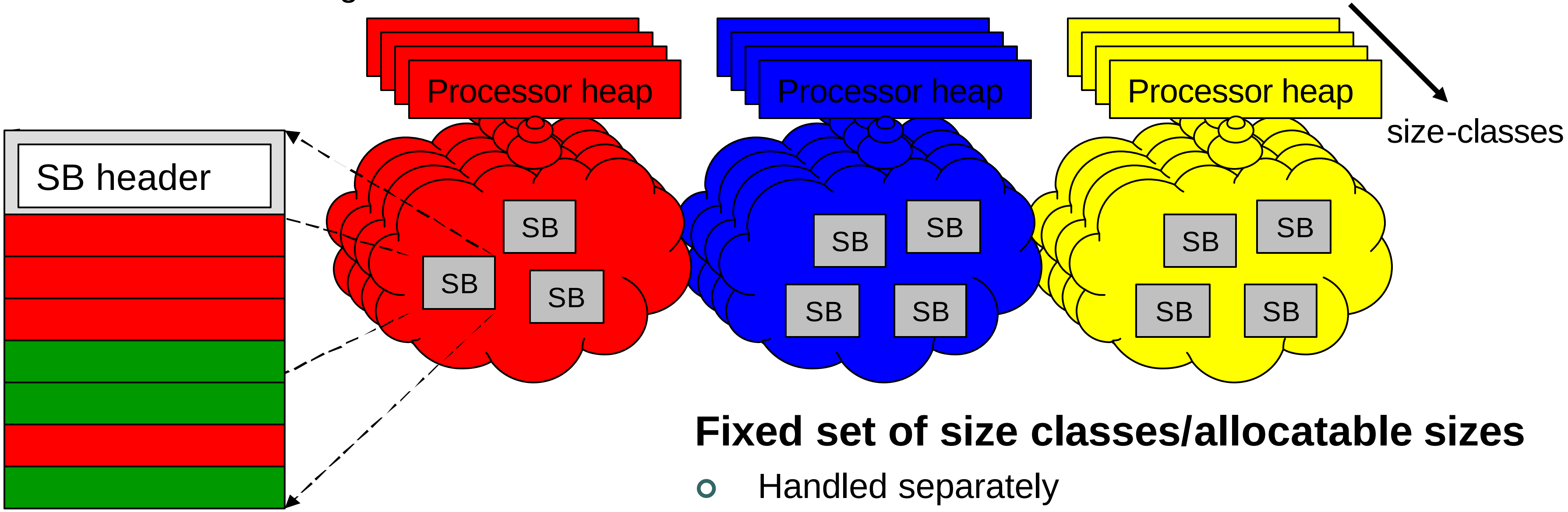
The Hoard architecture [Berger et al, 2000]

Superblocks

- Contains blocks of one size class
- Pros: Easy to transfer and reuse memory, prevents heap blowup
- Cons: External fragmentation

Per-processor heaps

- Threads running on different CPUs allocate from different places
- Avoids false-sharing and limits contention



Fixed set of size classes/allocatable sizes

- Handled separately
- Pros: Simple
- Cons: Increases internal fragmentation

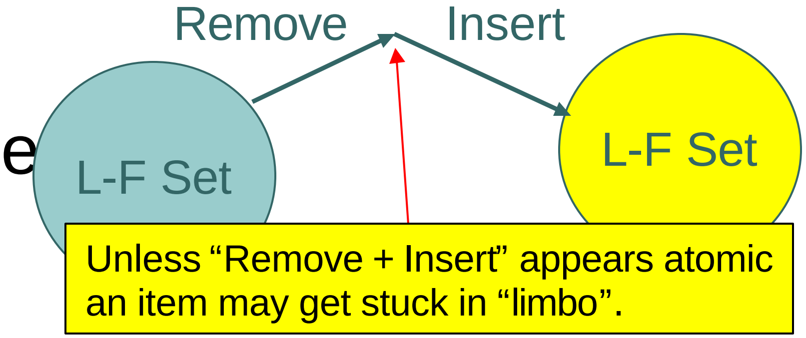
The lock-free challenges

1. The superblock internal freelist
    - Lock-free stack (a.k.a. IBM freelist [IBM, 1983])
2. Moving and finding superblocks within a per-processor heap
3. Returning superblocks to the global heap for reuse
    - New lock-free data structure: The flat-set.
        - Find an item in a set
        - Move an item between sets atomically

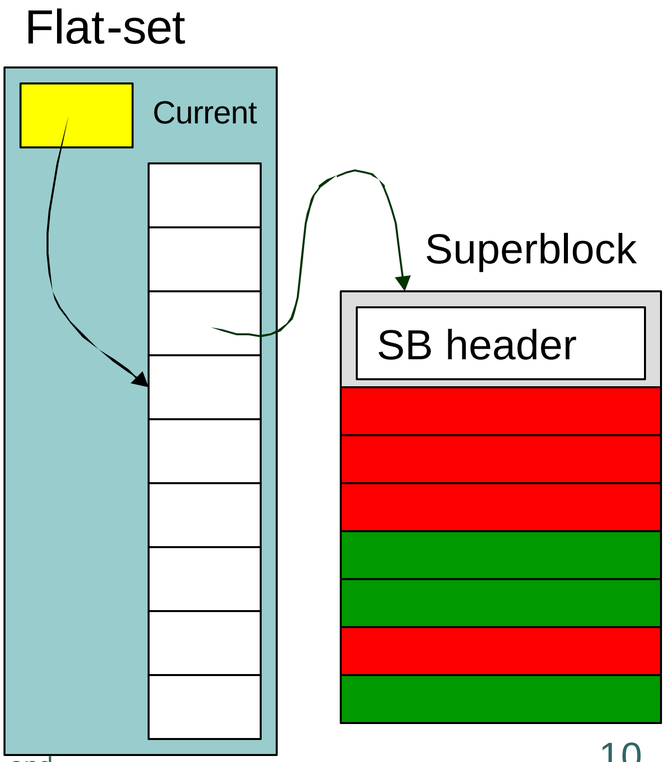
Lock-free flat-sets

Lock-free container data structure

- Properties
  - Items can be moved from one set to another atomically
  - An item can only be in one “set” at a time
- Operations
  - *Insert*
  - *Get_any*
  - *Insert* atomically removes the item from its old location



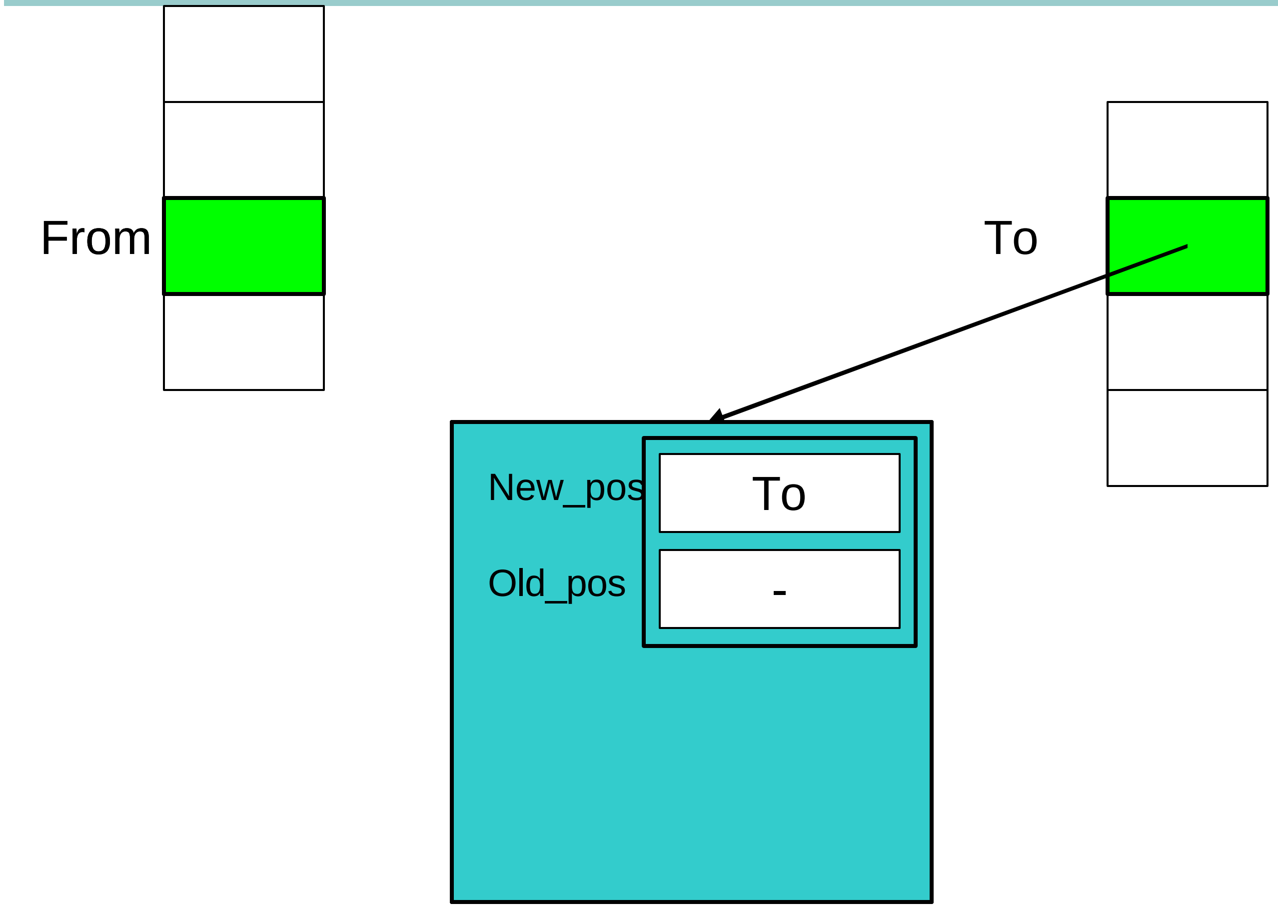
Unless “Remove + Insert” appears atomic an item may get stuck in “limbo”.



Moving a shared pointer

- Goal:
  - Move a pointer value between two shared pointer locations
- Requirements
  - The pointer target must stay accessible
  - The same # of shared pointers to the target after the move as before
  - Lock-free behaviour
- Issues
  - One atomic CAS is not enough! We'll need several steps.
  - Interfering threads need to *help* unfinished operations

Moving a shared pointer



Note that some extra details are needed to prevent ABA problems.

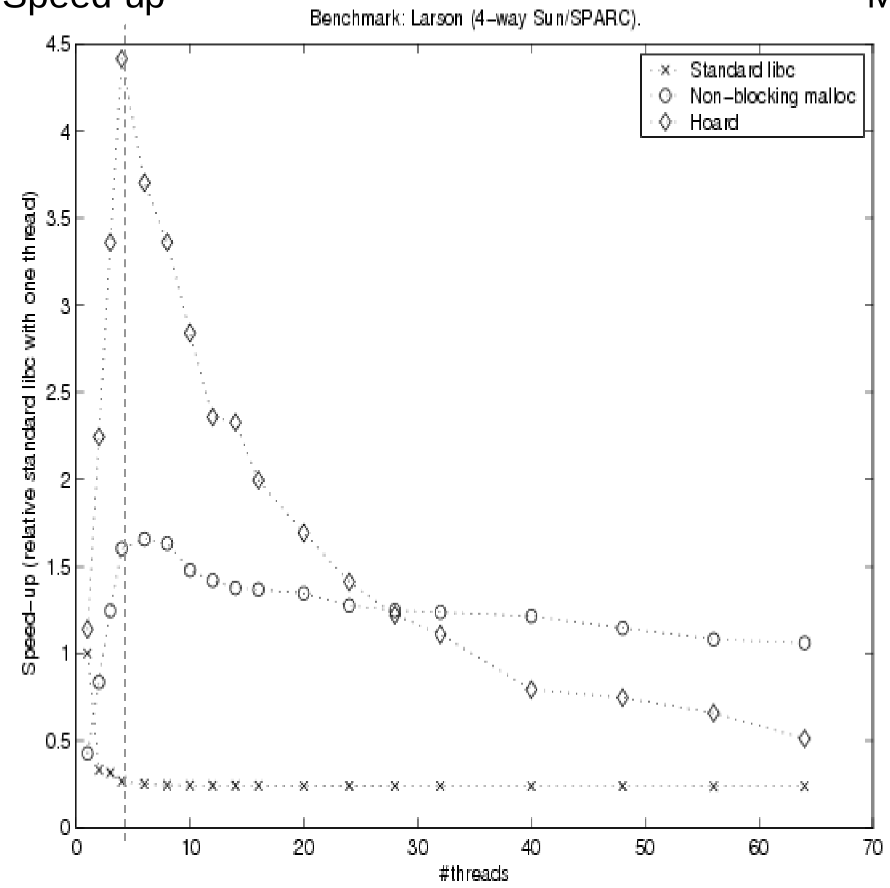
Experimental results

- Benchmark applications
  - Larson
    - Scalability
    - False-sharing

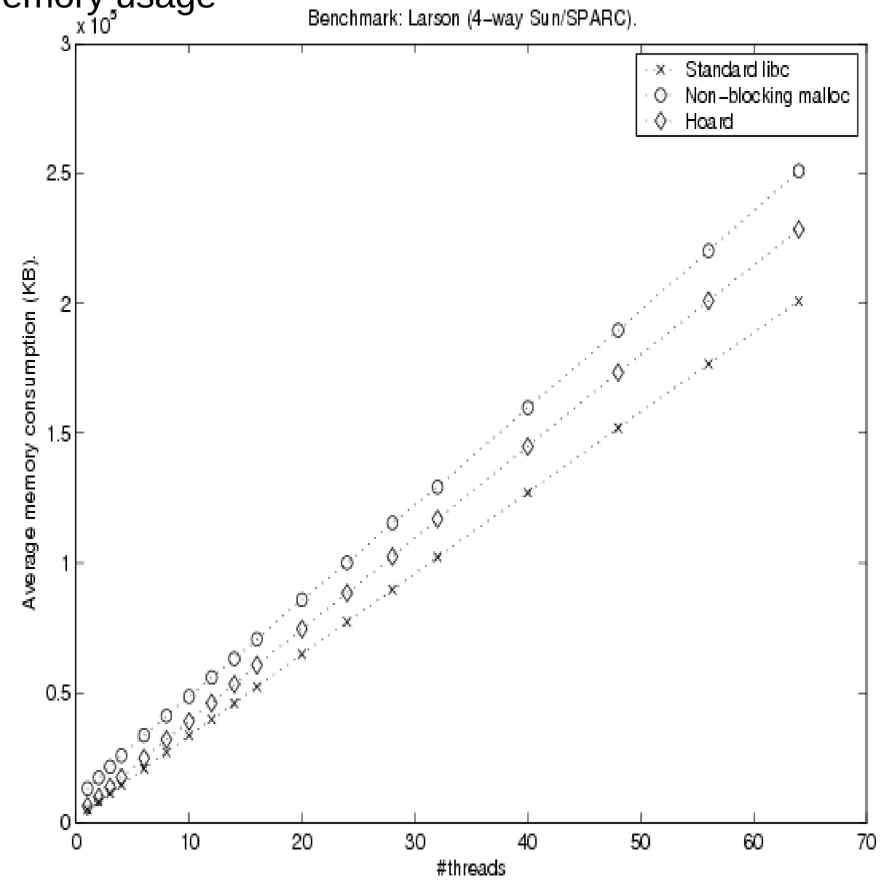
  - Active-false/Passive-false
    - Active false-sharing
    - Passive false-sharing

Experimental results

Speed-up

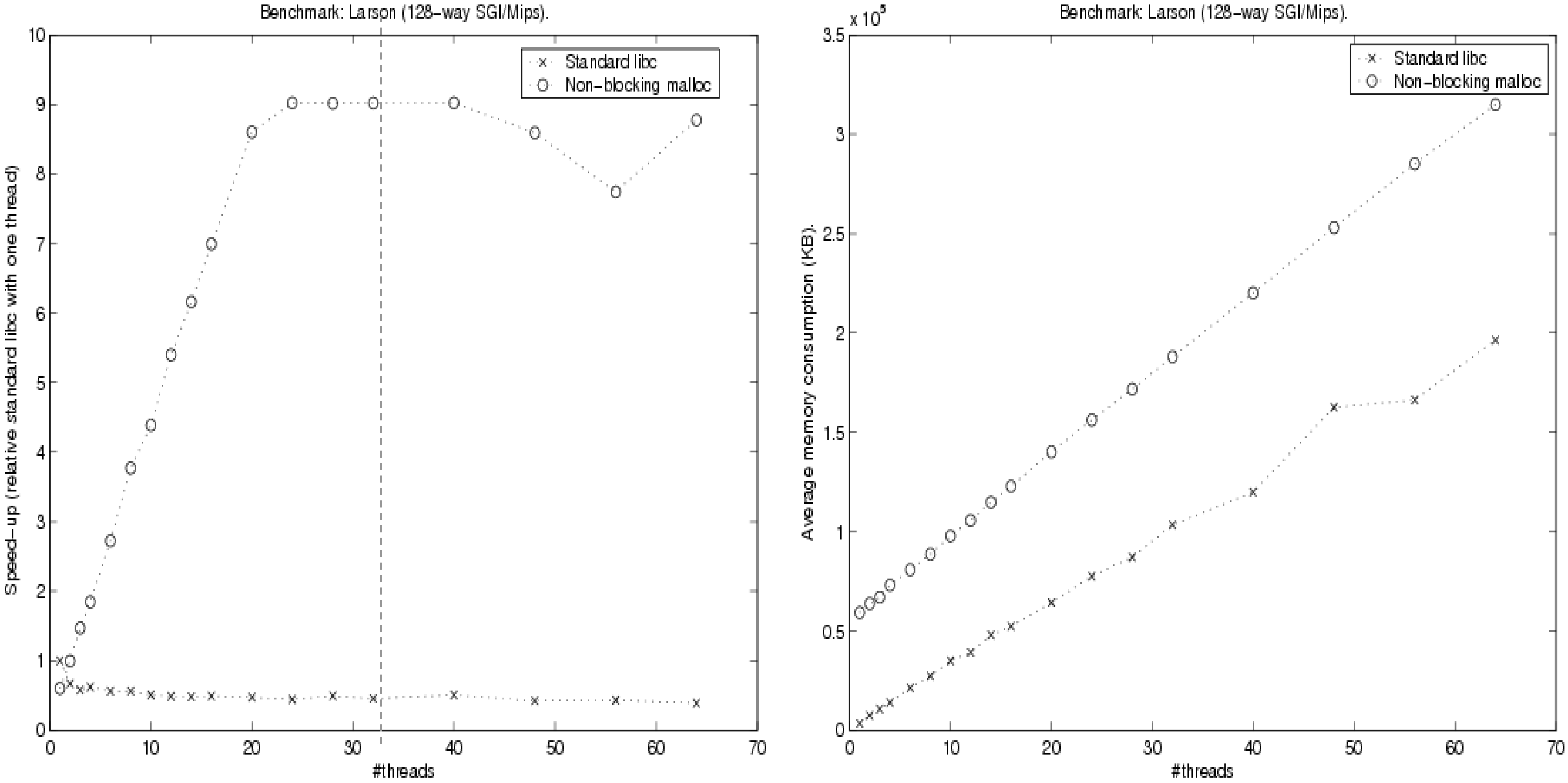


Memory usage



Larson benchmark. Sun 4xUltraSPARC III

Experimental results


Speed-up

Memory usage

Larson benchmark. SGI Origin 3800 32(/128)xMIPS

Conclusions

- **Lock-free memory allocator**
  - Scalable
  - Behaves well on both UMA and NUMA architectures
- **Lock-free flat-sets**
  - New lock-free data structure
  - Allows lock-free inter-object operations

- Implementation
  - Freely available (GPL)

Future Work

- Further development of the memory allocator
  - Reclaiming superblocks for reuse in a different size class
  - Improve search strategies for flat-sets
- Evaluate the memory allocator with real applications
- How to make lock-free composite objects from “smaller” lock-free objects

Questions?

- Contact Information:
  - Address:

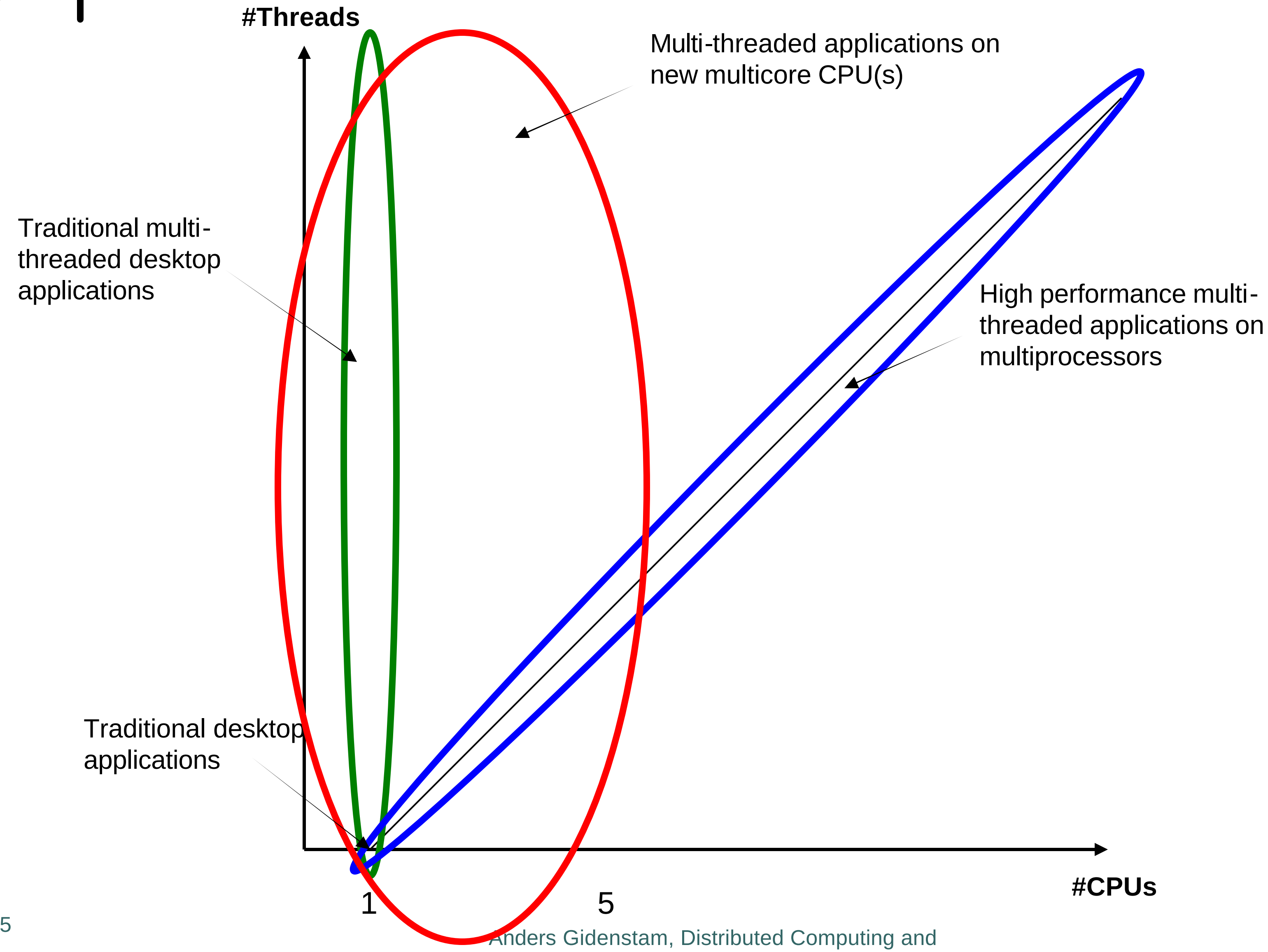
    Anders Gidenstam,
    Computer Science & Engineering,
    Chalmers University of Technology,
    SE-412 96 Göteborg, Sweden
  - Email:

    andersg @ cs.chalmers.se
  - Web:

    http://www.cs.chalmers.se/~dcs
    http://www.cs.chalmers.se/~andersg
  - Implementation

    http://www.cs.chalmers.se/~dcs/nbmalloc.html

Concurrent applications



Multi-threaded applications on new multicore CPU(s)

Traditional multi-threaded desktop applications

High performance multi-threaded applications on multiprocessors

Traditional desktop applications