Multi-Core and Embedded Software: Optimize Performance by Resolving Resource Contention

---

Presented by McObject LLC

February 29, 2012

mcobject
precision | data management

Achieving Linear Performance Gains With Multi-Core

- Multi-core CPUs should make software faster
- But, processes often contend for system resources
  - Threads vying for the standard C runtime memory allocator
  - Contention for shared data
- Solutions
  - Custom per-thread allocator
  - Multi-version concurrency control (MVCC)

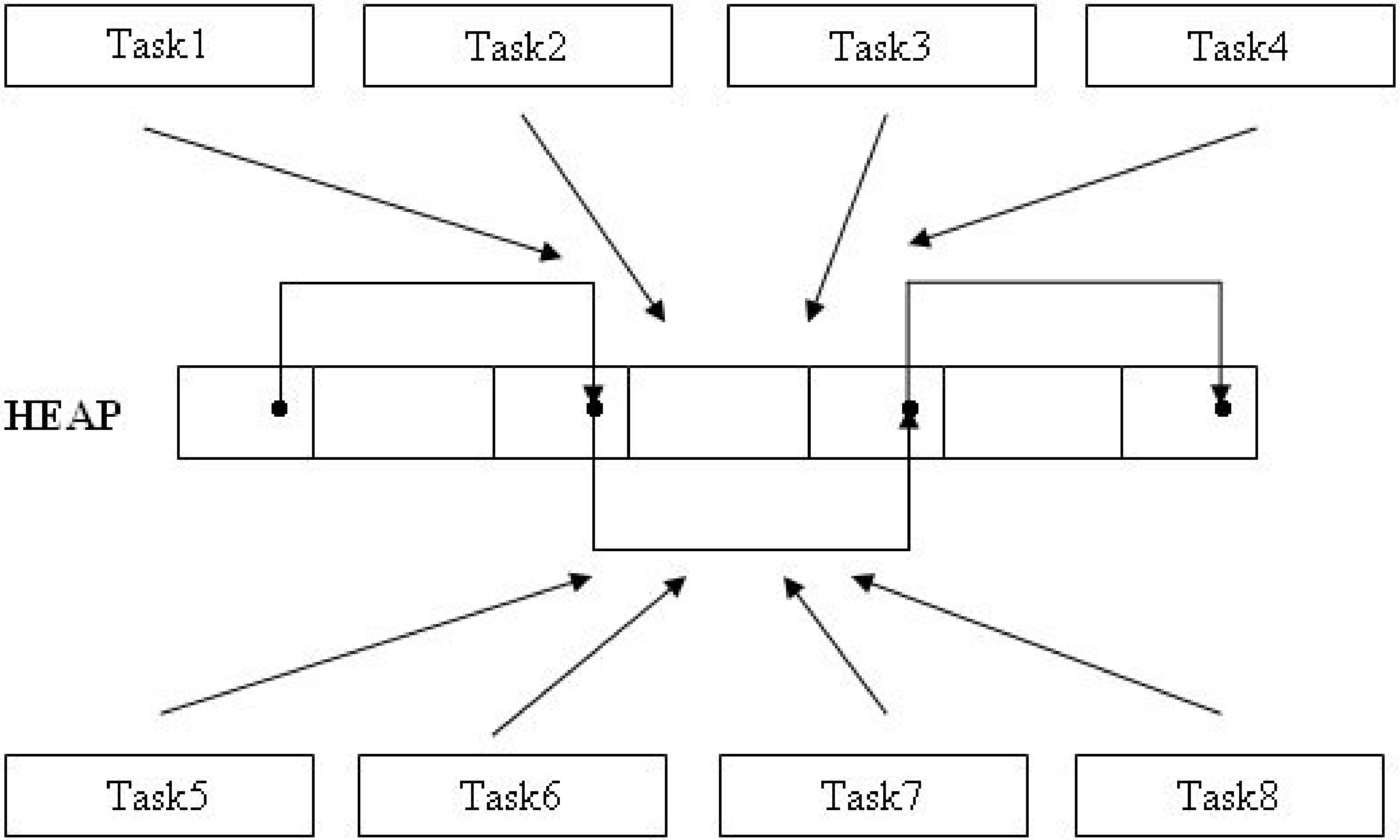
Memory Allocation

- malloc() and free()
- *new* and *delete*
- Used liberally
- But without awareness of how they actually work

The “Heap”

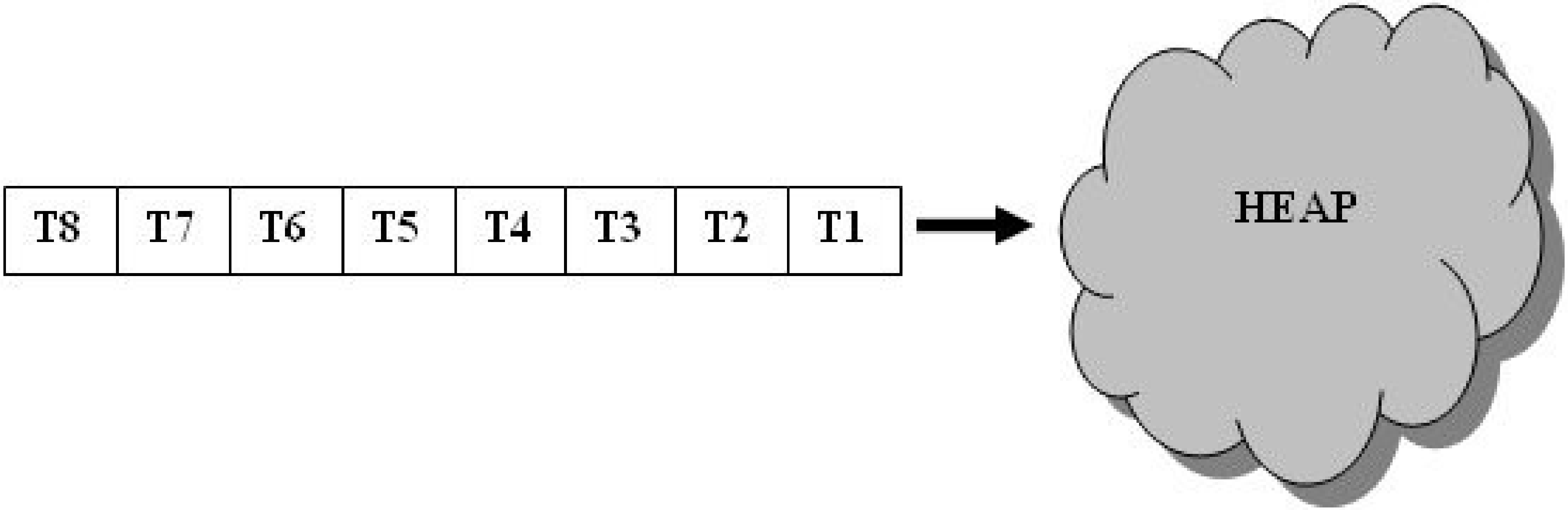
- Organized as a pool of contiguous memory locations
- Referenced by a singly-linked chain of pointers
- Memory allocation:
  - Walk the chain looking for a large enough free hole
  - When found
    - Unlink the hole
    - Divide it
    - Link remainder back in
    - Return pointer to allocated memory

Many Threads Want the Same Resource…



© 2012 McObject LLC

…End Up Being Serialized



The Solution

- A custom memory manager that avoids synchronization
- Thread local allocator
- Based on block allocator
- Similar concept to Thread Local Storage

Thread-Local Allocator

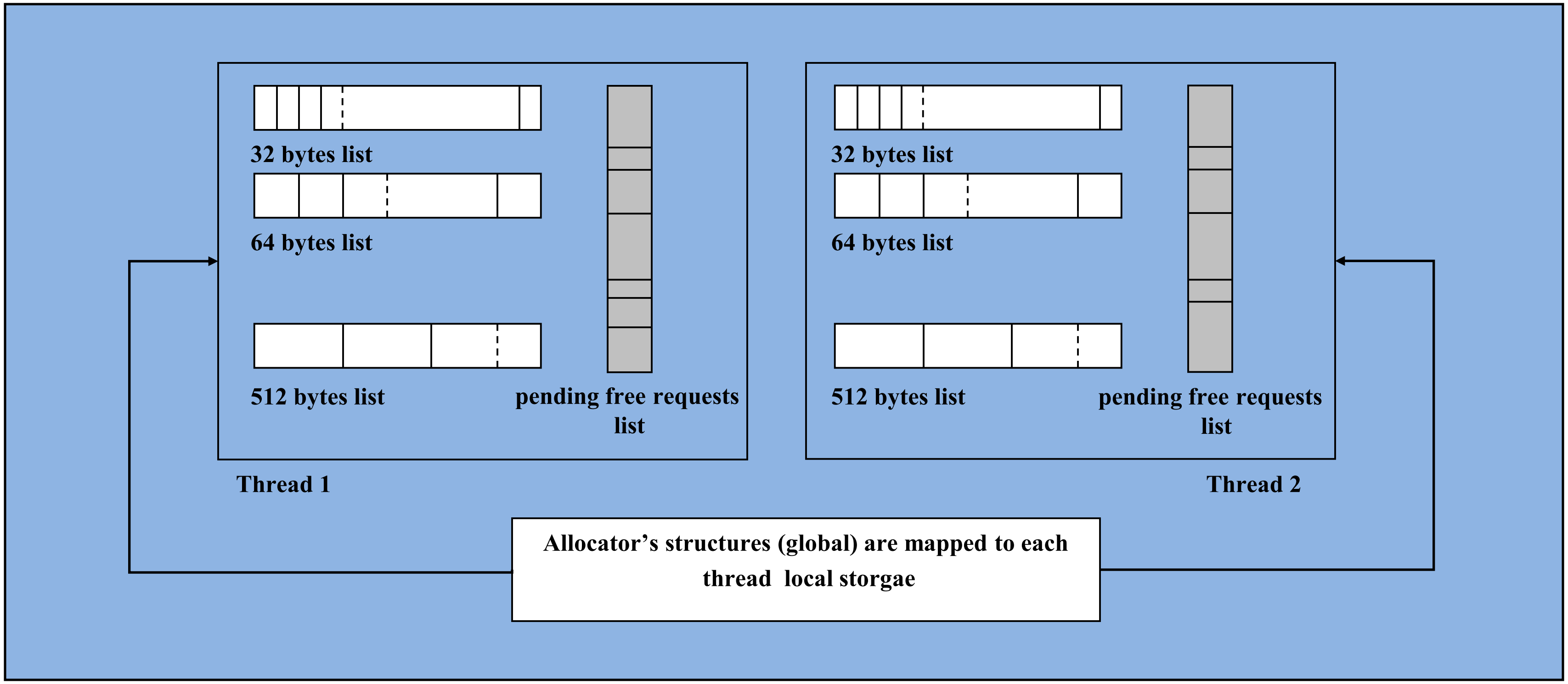
- Allocator creates and maintains a number of linked-lists (chains) of same-size “small” blocks that are made out of “large“ pages.
- To allocate memory, the allocator simply “unlinks” the block from the appropriate chain and returns the pointer to the block.
- When a new large page is necessary, the allocator uses a general-purpose memory manager (standard malloc) to allocate the page.
- As long as all objects are allocated and de-allocated locally (i.e. by the same thread), this algorithm does not require any synchronization mechanism at all.

Thread Local Allocator

- **Pending-free requests lists** (PRLs) are maintained for each thread: when an object allocated in one thread is being de-allocated by another thread, the de-allocating thread links the object into this list.
- Access to the PRLs is protected by a mutex.
- Each thread periodically de-allocates its share of objects on the list at once.
- **The number of synchronization requests is reduced significantly:**
  - Often the object is freed by the same thread that had allocated it.
  - When the object is de-allocated by a different thread, it does not interfere with all other threads, but only with those that need to use the same PRL.

mcobject®
precision | data management

Thread-Local Allocator Data Structures



Each thread's allocator maintains its "own" local data that includes the chains of blocks and its "pending free request list" within its TLS variables.

Allocator API

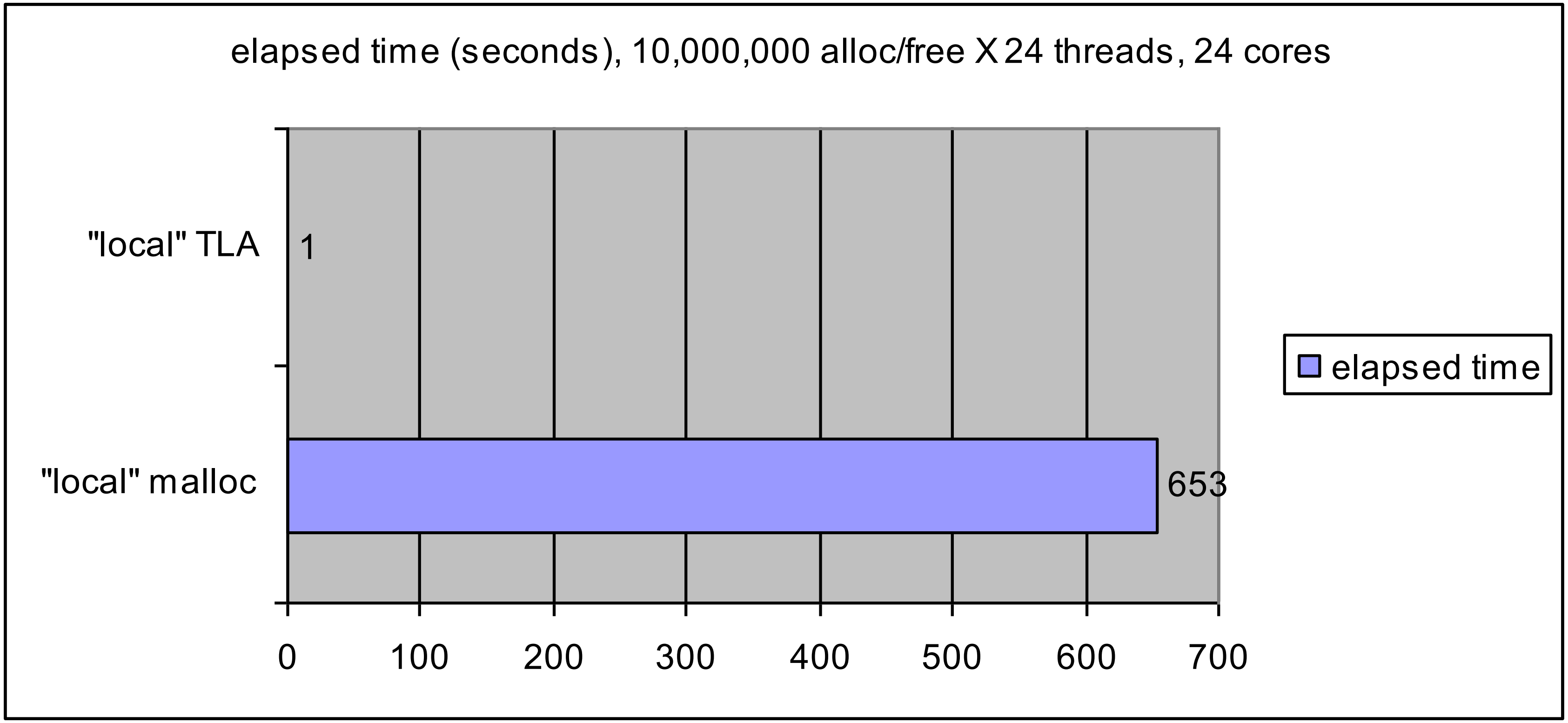
- The allocator exports three functions with syntax similar to the standard C runtime allocation.
- The interface also includes a simple way to redefine the default *new* and *delete* operators.

```
#ifndef __THREAD_ALLOC_H__
#define __THREAD_ALLOC_H__

#include <stddef.h>

#ifdef __cplusplus
extern "C" {
#endif

/* exported stuff */
void* thread_alloc(size_t size);
void* thread_realloc(void* addr, size_t size);
void  thread_free(void* addr);

#ifdef __cplusplus
}

/* redefine standard “new” and “delete” if necessary */
#include <new>

#ifdef REDEFINE_DEFAULT_NEW_OPERATOR

void* operator new (size_t size) throw(std::bad_alloc) { return thread_alloc(size); }
void operator delete (void* addr) throw() { thread_free(addr); }

void* operator new[](size_t size) throw(std::bad_alloc) { return thread_alloc(size); }
void operator delete[](void* addr) throw() { thread_free(addr); }

#endif
#endif
#endif
```

Impact of Thread Local Allocators

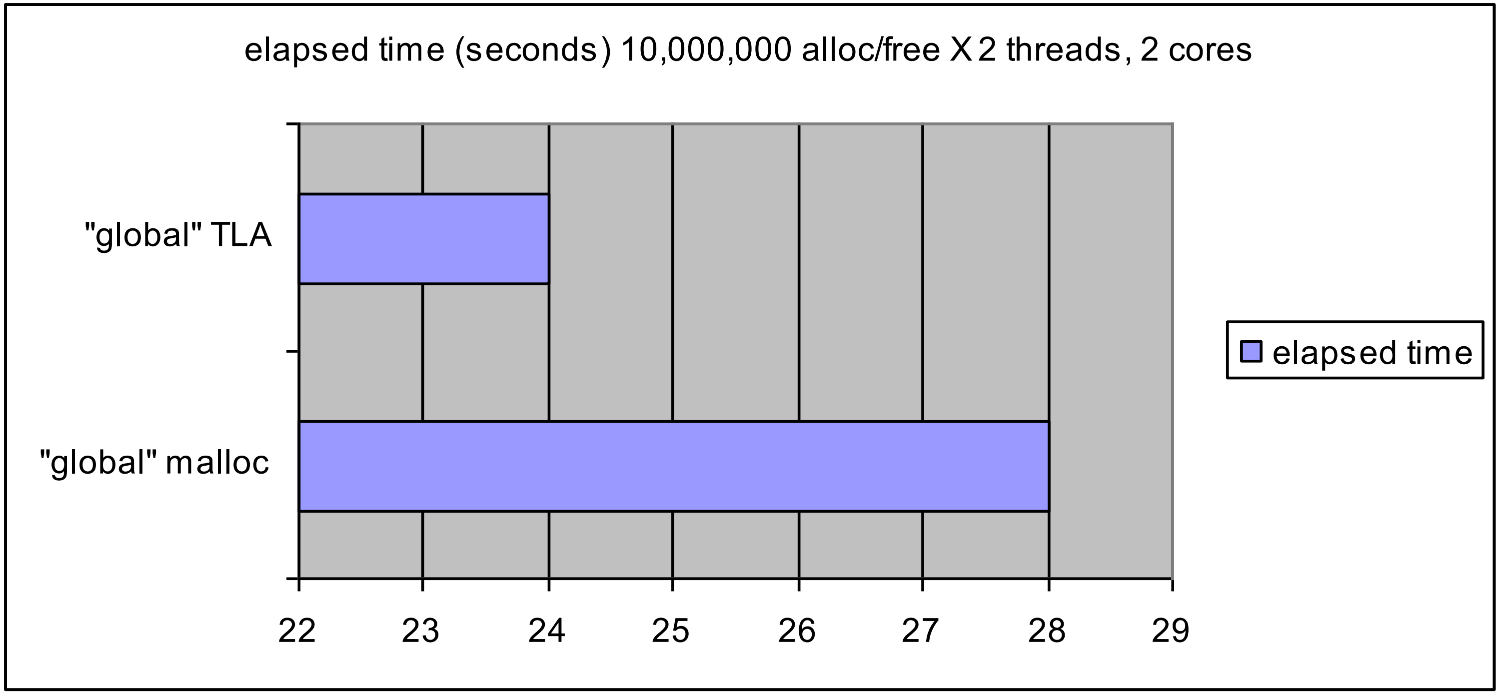
- Two tests
  - Compare performance when the allocation pattern is thread-local: all de-allocations are performed by the same thread as the original allocations.
  - Compare performance when objects are allocated by one thread (called a producer) and freed by another (a consumer).

Test Results



The graph depicts elapsed time when allocation and release of memory are both within the same thread.

Test Results, Cont.



The graph depicts elapsed time when every allocation is freed by a thread other than the one that allocated it. The test was run with just two threads to isolate the performance difference to just the reduced synchronization requirements of the thread-local allocator, even when all allocations are "global".

Test Results, Cont.

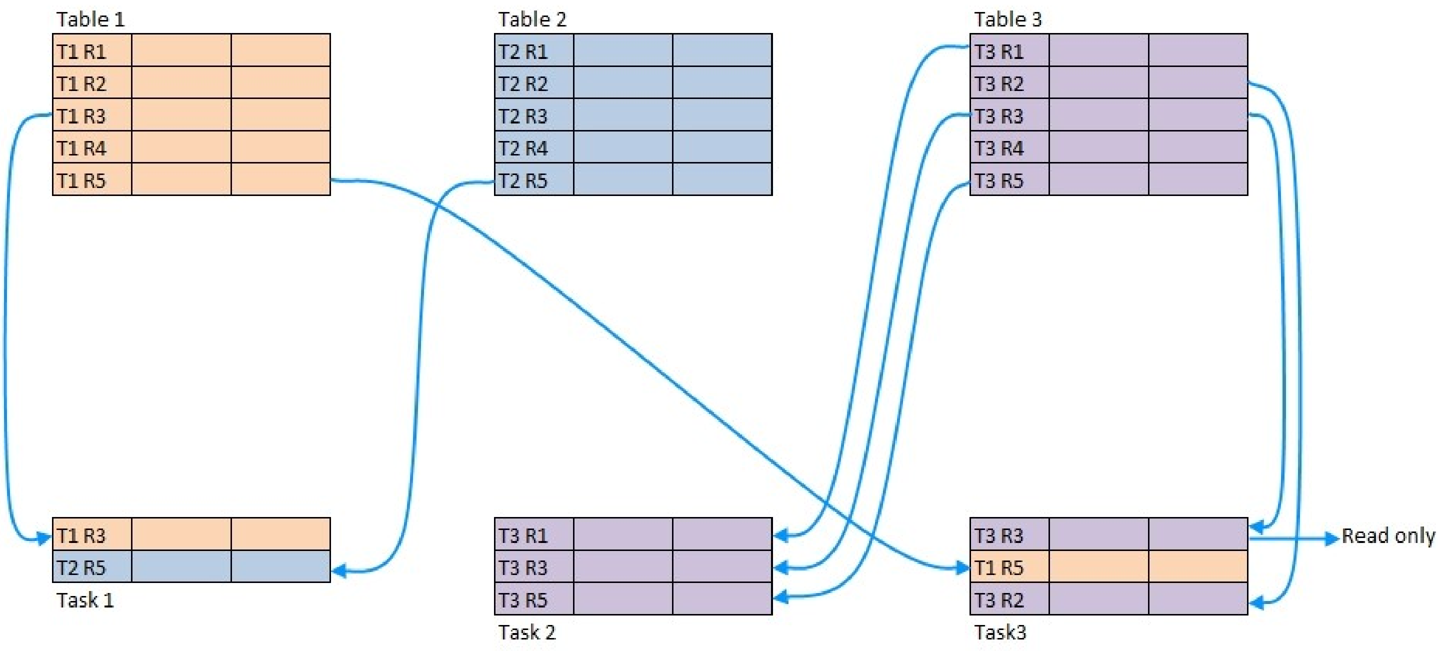
- Result: dramatic performance improvements are obtained by replacing standard allocation mechanism with thread-local in multi-threaded, multi-core applications.
- The allocator and test source code are available for free download: www.mcobject.com/webinar_mem_mgt

Contention For Shared Data

- Similar problem to memory allocation
- Shared resource must be protected
- Pessimistic locking is the norm
- Pessimistic locking blocks concurrent access (i.e. serializes) regardless of granularity

Pessimistic Locking

- Database
- Table
- Row



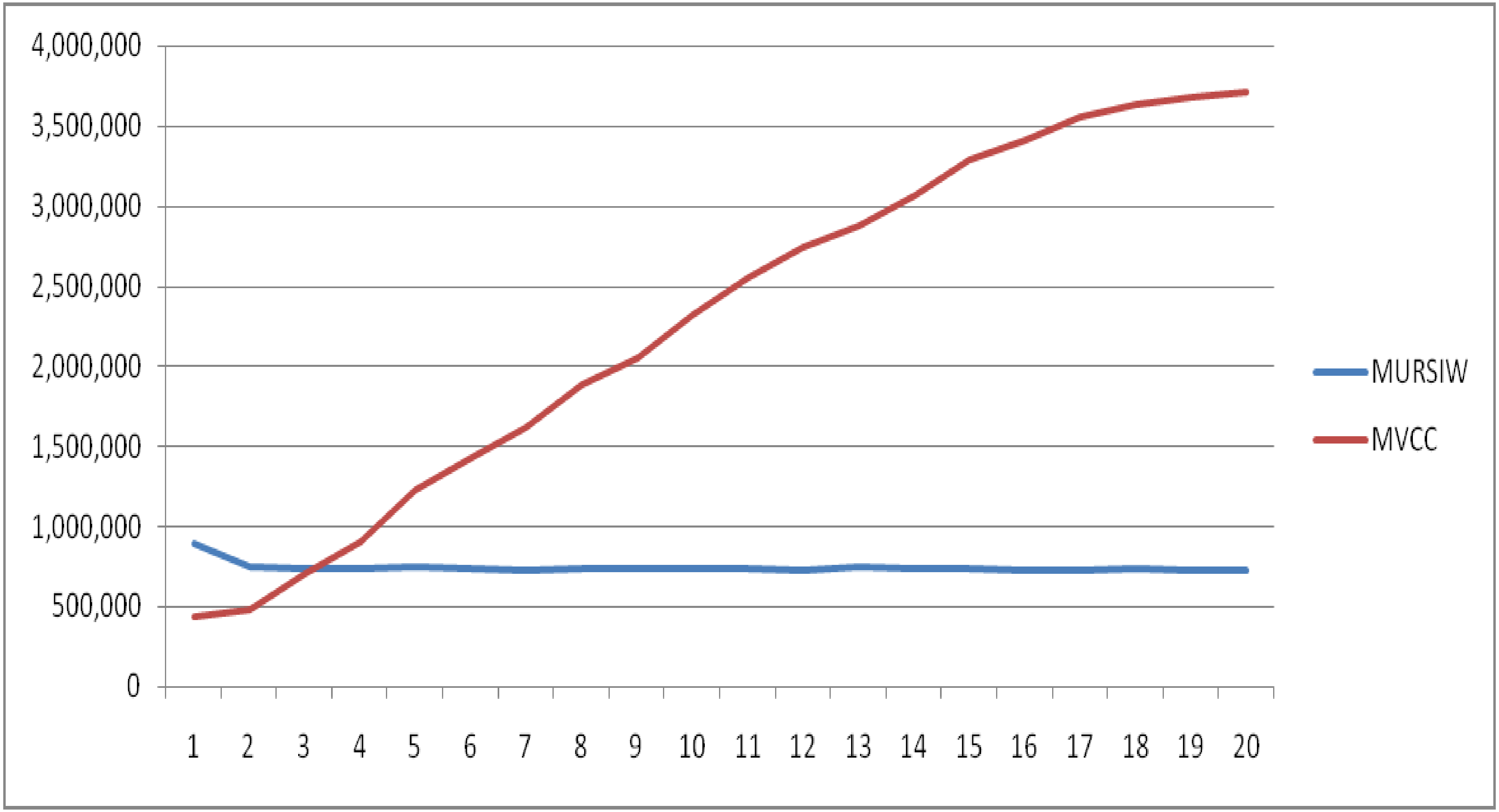
Pessimistic Locking

- Database locking
  - Read+write accesses are serialized
  - Read-only accesses are parallel with other read-only accesses, but blocked by read+write accesses
- Table locking
  - Read+write accesses by 2+ transactions that touch any common table are serialized
  - Read-only is parallel, but only between read+write accesses
- Row locking
  - Read+write accesses by 2+ transactions that touch any common row are serialized
  - Read-only is parallel, but only between read+write accesses

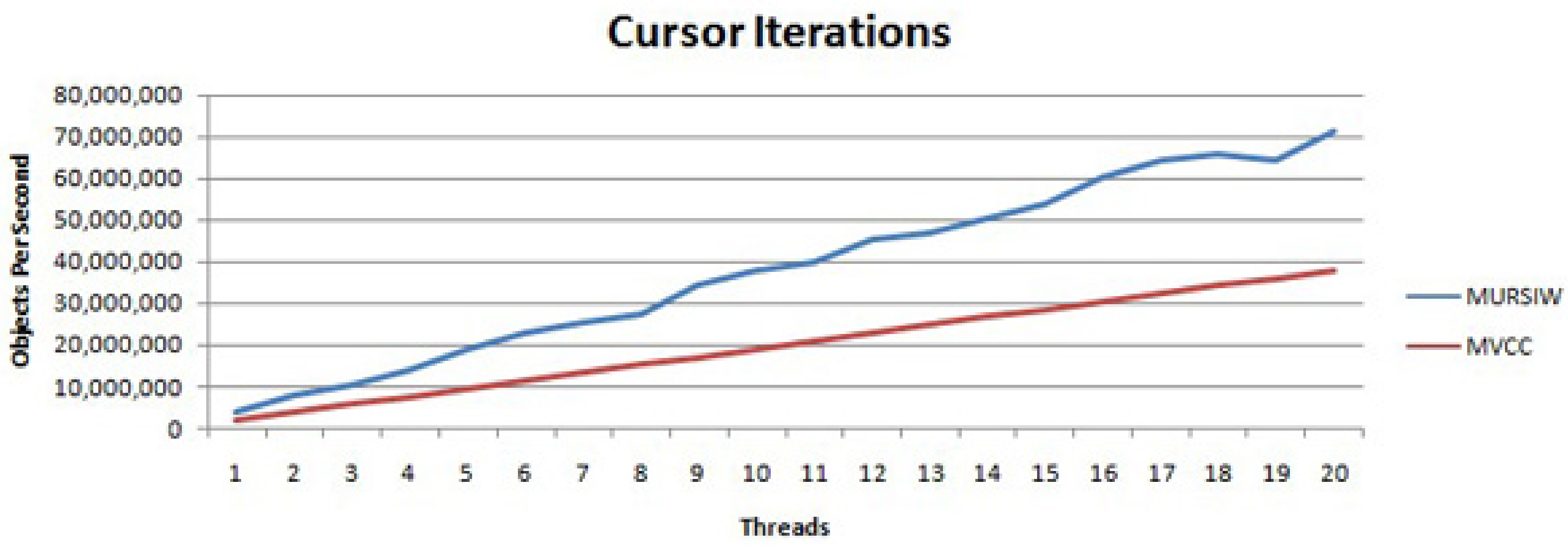
Multi-Version Concurrency Control (MVCC)

- Optimistic model, no locks & no complex lock arbitration
- No task is ever blocked by another
- Each task given a copy ("version") of objects it works with
- No serialization
- Similar in concept to
  - Read-Copy-Update (RCU) and
  - Software Transactional Memory (STM)

MVCC Update Performance vs. Database Lock



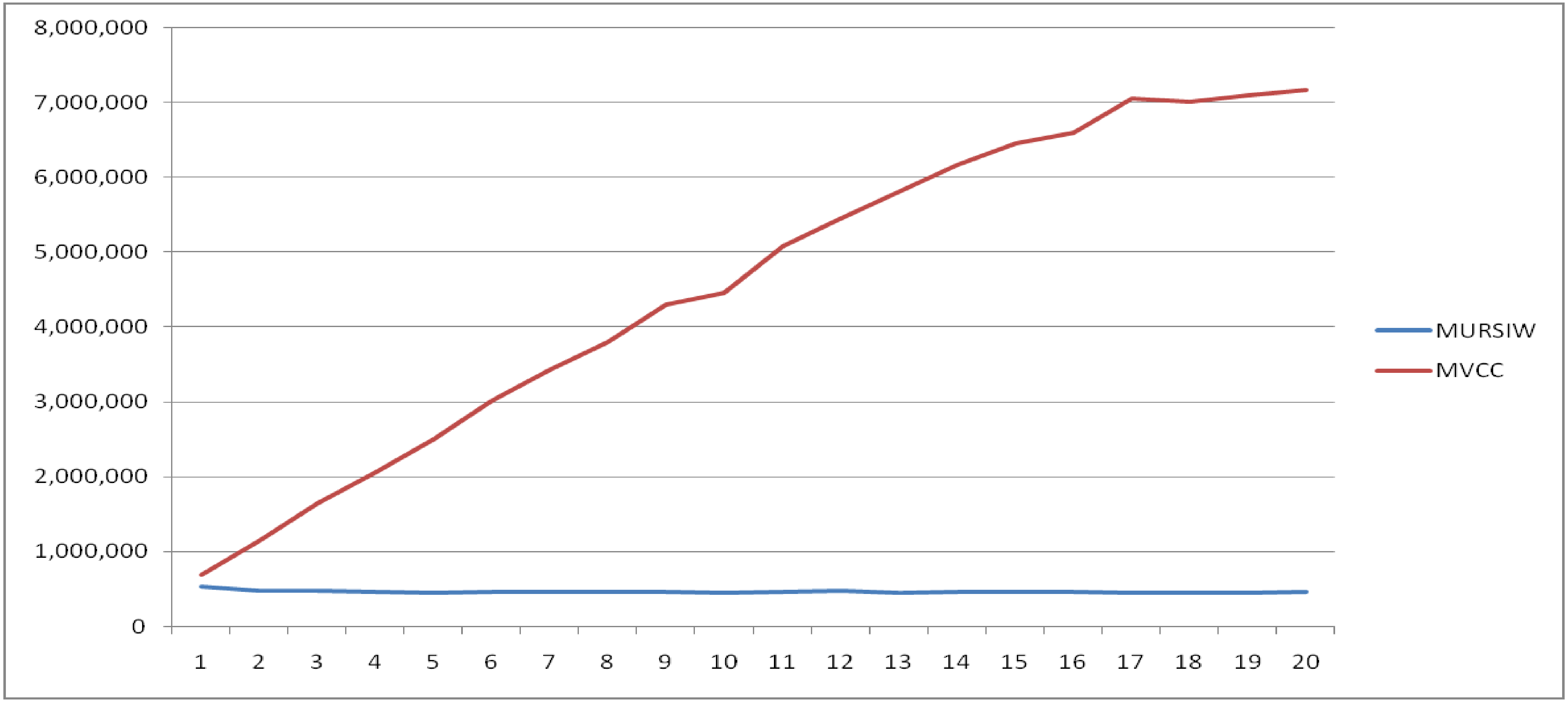
MVCC Read-Only Performance vs. Pessimistic Lock



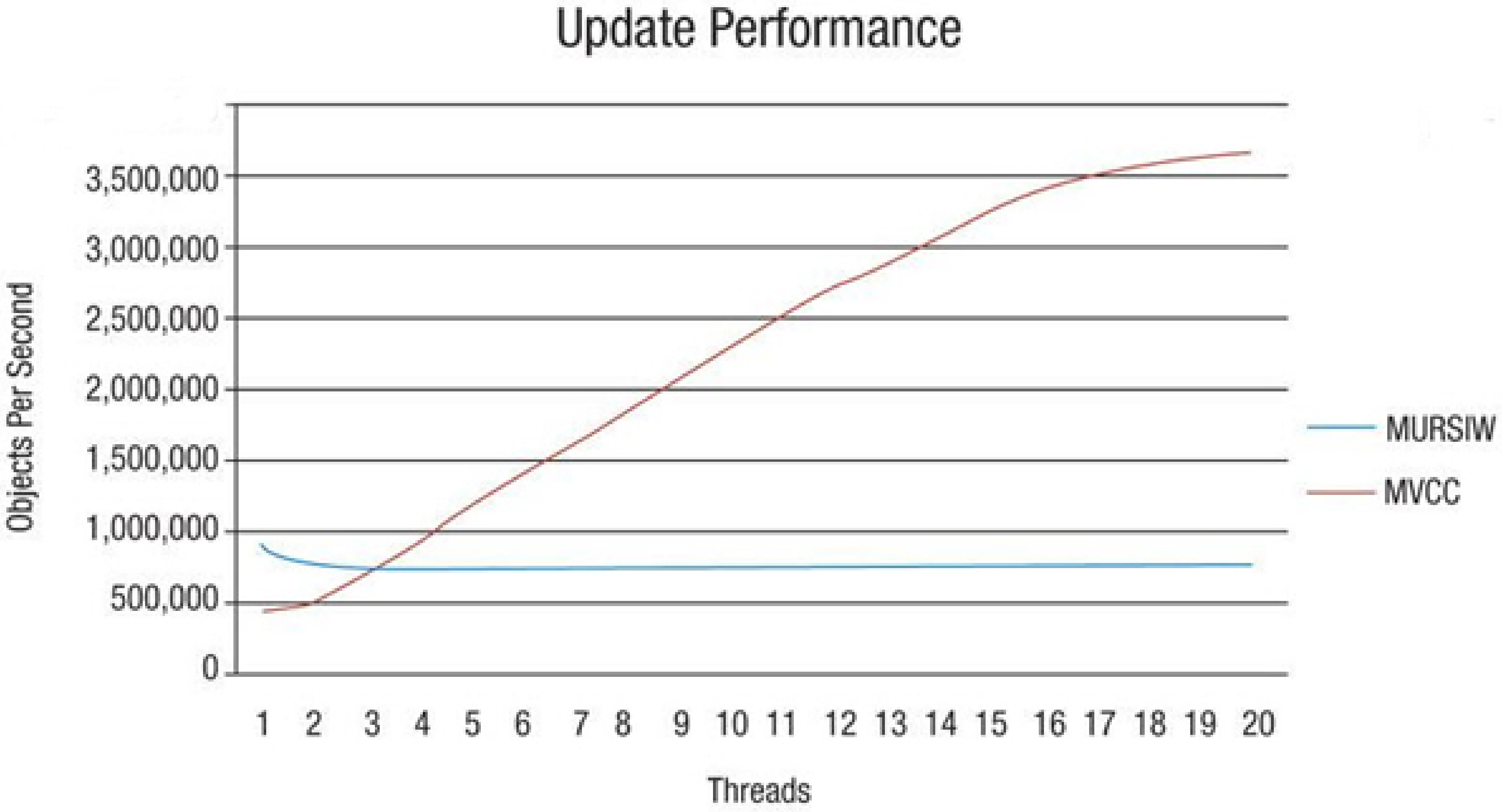
MVCC Insert Performance vs. Database Lock

Y-axis is objects acted on per second

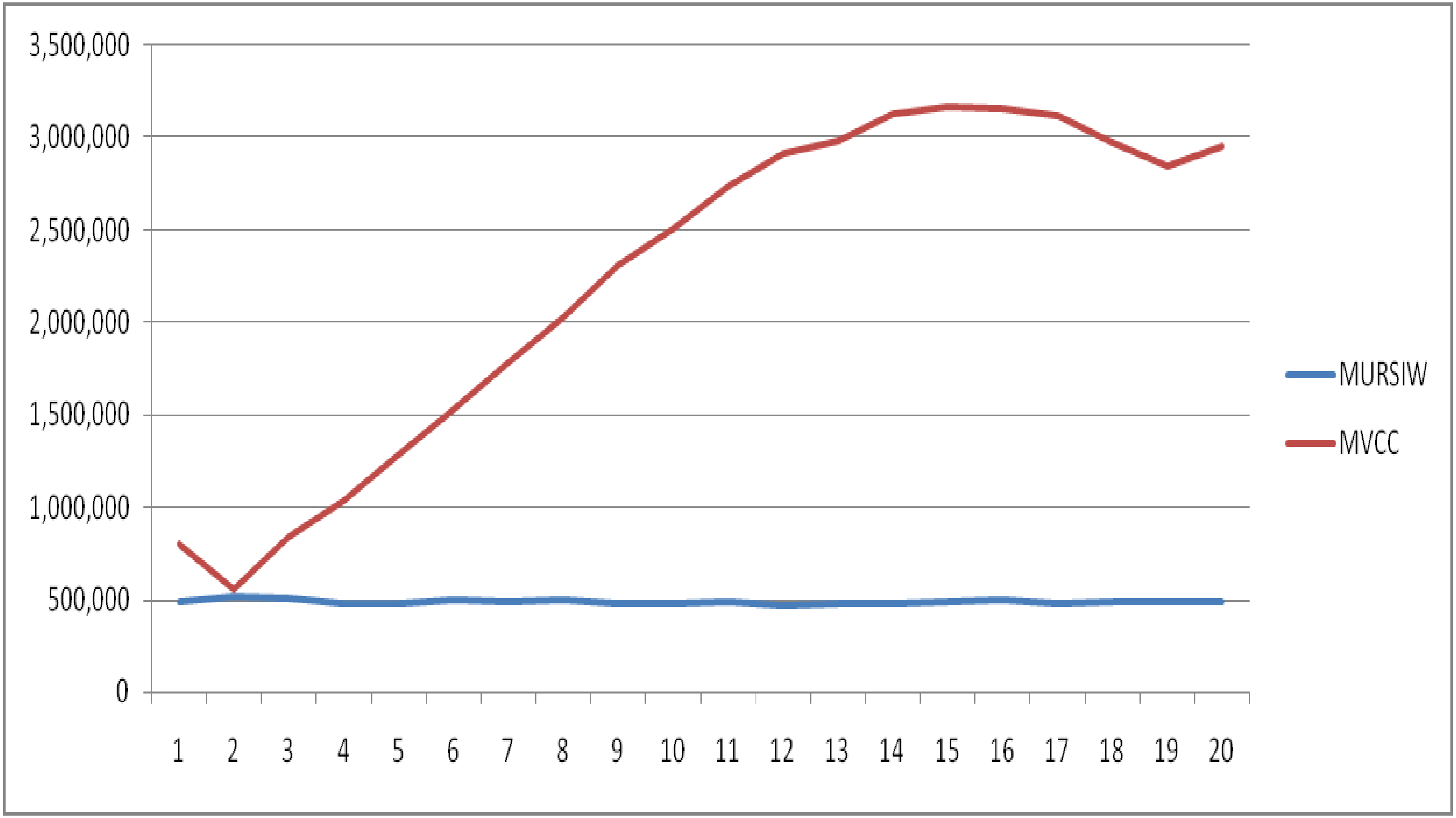
X-axis is number of cores



MVCC Update Performance vs. Database Lock



MVCC Delete Performance vs. Database Lock



Observations

- Pessimistic coarse-grained locking exhibits virtually no overhead
  - When access is read-only, scales very well
  - When access is read+write
    - Flat performance for multiple cores, i.e. it's N transactions whether there is 1 core or 8 cores
- MVCC/RCU/STM has greater overhead
  - Will never achieve equal read-only scalability
  - Needs some number of concurrent operations for concurrent ops to overcome greater overhead

Conclusions

- The goal is to maximize efficient use of multiple cores
- Be wary of how “black box” software components can work against your goal
- Case-in-point: malloc and free & pessimistic locking generally
- Old “single-core” methods don’t scale. Learn to think with a multi-core mentality