Paper title: Hoard: A Scalable Memory Allocator for Multithreaded Applications
Paper authors: Emery Berger, Kathryn McKinley, Robert Blumofe, Paul Wilson
Your name: Parth Kansara (115135130)

What problem does the paper address? How does it relate to and improve upon previous work in its domain?
Previous memory allocators perform and scale poorly on multiprocessor systems, along with introducing false sharing and unbounded memory consumption. They are lacking in one or more of the desired features, namely speed, scalability, false sharing avoidance and low fragmentation. Two important issues addressed by the paper are false sharing and blowup. False sharing is caused when multiple processors share the same cache lines while not sharing any data. Memory allocators may actively or passively induce this false sharing, which makes it a significant issue. Secondly, the problem of fragmentation is aggravated by the occurrence of *blowup* - defined as the failure of the concurrent allocator to reuse the freed up memory, thereby inflating the memory requirement of the application in an unbounded manner. This usually occurs in producer-consumer type of threads.

Serial single heap allocators such as the ones provided by **Solaris** and **IRIX** do not scale with multithreaded programs, since they use locking which causes contention. **Windows NT/2000** uses freelists instead of locks, but even these do not scale. These and the concurrent single heap allocators both actively induce false sharing. The latter also uses multiple locks or atomic update operations which is costly.

Pure private heap allocation, as in **STL**, **Cilk 4.1** and others, may place segments of the same cache line on different heaps, which passively induces false sharing, along with bearing the problem of memory blowup. Private heaps with ownership, as used by **MTmalloc, Ptmalloc** and **LKmalloc**, result in O(P) blowup, which is still underperforming as compared to the paper's algorithm which bounds the blowup to O(1). Private heaps with threshold, as demonstrated by the **DYNIX kernel memory allocator** and by the allocator used by **Vee and Hsu**, are efficient and scalable, but still suffer from the issue of passively induced false sharing, since the same cache line may be reused. They are also subjected to the overhead of synchronization on every memory allocation or free up.

What are the key contributions of the paper?
The paper addresses the above issues and offers a scalable memory allocator, Hoard which resolves the issues of false sharing and blowups. It maintains heaps for every processor, along with a global heap, which gets allocated heaps from the processor when it crosses the emptiness threshold. The superblocks which constitute the processor heaps are ordered by a move-to-front heuristic, which maintains a LIFO order and establishes locality. This constrains the blowup by a constant factor and limits the synchronization overhead.

Also, simultaneous requests from multiple threads are serviced from different superblocks, which avoids actively induced false sharing. Upon deallocation of a block, Hoard returns it to the superblock which eliminates passively induced false sharing. These algorithms, upon thorough evaluation, prove that Hoard is able to bound blowup, reduce synchronization cost, reduce fragmentation, avoid false sharing and scale efficiently.

Briefly describe how the paper's experimental methodology supports the paper's conclusions.
In terms of speed, Hoard performs better than the Solaris allocator for applications that have high memory allocation. Hoard also scales linearly and outperforms Solaris and MTmalloc, which display a significant slowdown. Hoard also runs 278% faster than PTmalloc for 14 processors. On the active-false benchmark, Hoard outperforms both PTmalloc and MTmalloc while Solaris does not scale at all. On the passive-false benchmark as well, Hoard outperforms the other two, showing only a slight slowdown after 12 processors. It was also observed that the superblock acquired by the processor from the global heap was empty, again proving that Hoard eliminates any allocator-induced false sharing.

For single-threaded applications, Hoard resulted in very low fragmentation of around 1.05 to 1.2, except in the case of *Espresso.* For multi-threaded applications, Hoard resulted in nearly no fragmentation in certain cases. For *shbench*, Hoard's choice of keeping one size class per superblock resulted in poor memory performance, but this kind of application behavior is rare. In terms of sensitivity as well, Hoard was able to robustly maintain its runtime. These evaluations uphold the claims of Hoard's superior performance as compared to the existing memory allocators.

Write down one question you may want to ask the authors.
What could be some potential roadblocks in integrating Hoard into existing systems, as it requires the allocation of a global heap and the maintenance of usage statistics for the processor heaps?