Design And Implementation Of A Parallel Memory Allocator

Eric Bannatyne Kevan Hollbach

1 Introduction

As the needs of large-scale multithreaded applications increase, the mechanism by which memory is allocated for use by applications must scale accordingly, so as to not act as a bottleneck that may lead to poor scalability characteristics. Simply using a serial memory allocator in a multithreaded shared-memory system does not suffice, as increasing the number of threads that can execute concurrently will often lead to significant contention on memory operations. Therefore, it is necessary to design and implement a memory allocator for such tasks with scalability in mind. Our goal is to build a parallel memory allocator that executes memory operations (allocations and deallocations) quickly, with a minimal amount of overhead, and scales well as the number of processors increases, while avoiding false sharing of cache lines and minimizing fragmentation.

2 Design

Our parallel memory allocator implementation primarily draws ideas from Hoard [1]. The main idea is that every CPU has its own private heap, which is composed of a number of subheaps, each corresponding to a single size class. Each subheap contains an array of linked lists of superblocks, which correspond to different fullness bins, which provides a number of performance improvements in the implementation of `mm_malloc` and `mm_free`. There is also a global heap that provides superblocks that can be used by any thread.

In addition to directly implementing the pseudocode of Hoard, we included a number of additional mechanisms that would be useful. For example, something that is only briefly alluded to (but not actually in the pseudocode) in the original Hoard paper is the recycling of used superblocks. In particular, once a superblock becomes empty, it can be reused as a superblock for a different size class. Thus we decided to include a list of empty blocks for every CPU heap (as well as the global heap), with a reclamation procedure designed specifically for empty blocks; since empty blocks do not have size classes, they are not directly affected by the regular superblock reclamation procedure, and must therefore be reclaimed separately.

Hoard includes a number of parameters that can be tuned to adjust the parallel memory allocator's performance characteristics. In our implementation, we had a superblock size of $S = 1$KB, with size classes of 8, 16, 32, 64, 128, 256, and 450 bytes. We chose to use a K threshold of $K = 4$. We used four equally-spaced fullness bins, which, for the sake of simplicity and efficiency, gives rise to an f-value of $f = 1/4$.

2.1 Synchronization

One of the challenges that we faced involved dealing with increased contention on heap locks when scaling to a larger number of CPUs. We initially implemented our memory allocator with a single lock per CPU heap, as well as one for the global heap. However, this led to a high degree of contention on these locks, slowing down the `mm_malloc` and `mm_free` operations significantly. Having mutual exclusion be handled at a subheap level (thereby allowing memory operations within the same CPU heaps but for different size classes to execute concurrently) led to significant performance improvements, including a tenfold increase in our scalability scores, especially for the Larson benchmark, which makes use of a variety of size classes.

One important aspect of our approach to synchronization is ensuring that deadlocks will not occur. This requires careful sequences of locking and unlocking operations, ensuring, for example, that a thread never tries to acquire a subheap lock while holding any other lock.

An interesting scenario occurs at the start of `mm_free`, when the memory allocator needs to determine which subheap the current superblock belongs to, and acquire

its lock. In particular, it is necessary to attempt to acquire the lock in a loop, to avoid potential race conditions which may result from freeing a block from a superblock while another thread is in the process of moving it from one linked list to another. In this case, it is important to take care to ensure that the superblock's bin pointer is non-null and has not been changed since it was last accessed. In order for this code to work, it is important to use a compile-time memory barrier to prevent compiler optimizations from reordering certain lines of code, which, if not accounted for, leads to situations in which a thread can hang indefinitely, without making any progress.

2.2 Performance Optimizations

By far, the largest performance and scalability improvement that we gained was due to replacing per-heap mutexes with per-size class mutexes, which led to a significant decrease in lock contention. Switching to more fine-grained locking patterns, thereby minimizing the amount of time a thread spends holding onto a given lock, yielded further performance increases. Beyond that, we used the profiling tool `gprof` to determine where the most time was being spent in our code, in order to find potential optimizations. One simple optimization that yielded significant benefits was to precompute frequently-used values, such as the number of blocks that can fit into a superblock of a given size class. Similarly, we ensured that the memory allocated for CPU heaps was aligned with the system's CPU cache line size (64 bytes) in order to avoid false sharing when threads access their private CPU heaps.

One interesting finding was that `gprof` seemed to indicate that our code was spending a significant time in the function `get_size_class`, which simply finds the appropriate size class for a given memory allocation request, and is called once for every call to `mm_malloc`. This function had originally been implemented as a simple `for` loop that was a surprising bottleneck in our code. Moreover, based on our analysis, we found that, in the cache-scratch, cache-thrash and threadtest benchmarks, almost all memory allocation requests went to the 8-byte size class, whereas for the Larson benchmark most `mm_malloc` requests were concentrated within the 32-byte size class, with the next most common size classes being 16 and 64 bytes. Size classes larger than 64 were rarely used in practice in the benchmarks. Therefore, we decided to optimize `get_size_class` by manually implementing an unbalanced decision tree designed to minimize the number of CPU cycles needed to categorize allocation requests into the 32, 8, 64 and 16-byte size classes, in that order. Based on our profiling results, this, surprisingly, led to a nontrivial performance improvement in our `mm_malloc` implementation.

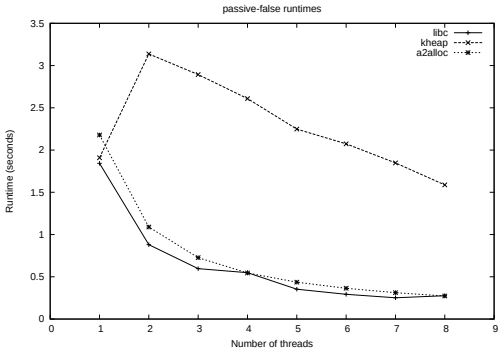
Throughout the process of developing our implementation of a2alloc, we experimented with a number of parameter settings, including superblock size. As we experimented with different values for the superblock size parameter, we found that there was a tradeoff between speed and fragmentation resulting from different superblock sizes. In particular, decreasing the superblock size tended to decrease fragmentation, particularly for benchmarks that, when run with a larger superblock size, did not actually require more than one superblock. Moreover, when we tried using smaller superblocks, we also reduced the number of size classes available (to ensure that every superblock can have at least two blocks), which reduced fragmentation simply by avoiding allocating memory for subheap headers that never actually get used by the benchmarks. However, when the superblock size is reduced too much, this also led to a decrease in sequential speed, due to increased time spent moving superblocks between heaps. Based on our experiments, we found that choosing a superblock size of $S = 1$KB yielded the best balance between fragmentation and speed.

3 Results

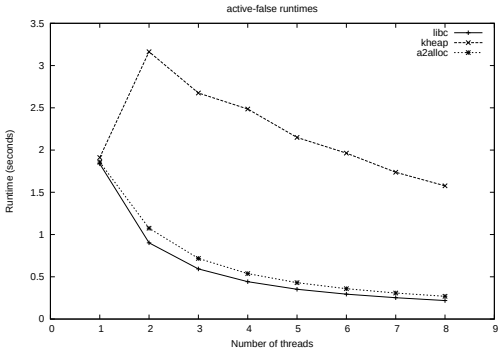
Our results from running the benchmarks on the timing server are as follows.

```
cache-scratch/Results/a2alloc/scores
        sequential speed    = 0.866
        scalability score   = 0.999
        fragmentation score = 0.298

cache-thrash/Results/a2alloc/scores
        sequential speed    = 1.017
        scalability score   = 0.864
        fragmentation score = 0.326

larson/Results/a2alloc/scores
        sequential speed    = 0.675
        scalability score   = 0.527
        fragmentation score = 0.387

threadtest/Results/a2alloc/scores
        sequential speed    = 0.815
        scalability score   = 0.994
        fragmentation score = 0.216
```

From Figures 1a and 1b, we see that our implementation of a2alloc scales comparably to the libc implementation, indicating that our implementation avoids both active and passive false sharing.

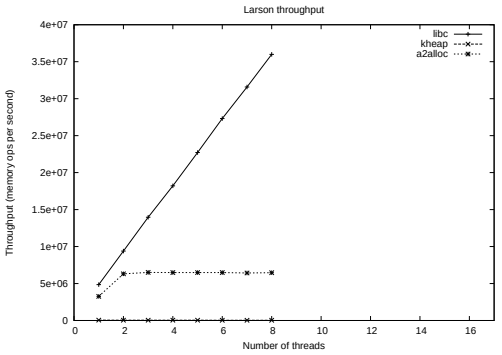


(a) Scalability of cache-scratch benchmark.

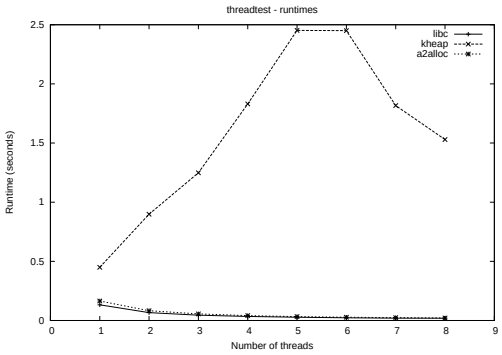


(b) Scalability of cache-thrash benchmark.

Figure 1: Scalability results for cache-scratch and cache-thrash benchmarks.



(a) Scalability of Larson benchmark.



(b) Scalability of threadtest benchmark.

Figure 2: Scalability results for Larson and threadtest benchmarks.

For the Larson benchmark, in Figure 2a, it is interesting to note that, while a2alloc performs better with two threads than one, beyond two threads throughput does not seem to increase. We conjecture that this may be due to lock contention, leading to diminishing returns when we have more than two threads executing concurrently. This hypothesis seems to be supported by the results of profiling our code, as run on the Larson benchmark, with Valgrind DRD, which indicated that, rather frequently, threads would often spend a significant amount of time (more than 10ms) waiting for mutexes. It would be interesting to try to modify our allocator in a way that reduces contention on these locks while retaining its fragmentation and scalability characteristics on other benchmarks. For threadtest, in Figure 2b, we see that the runtime for a2alloc does indeed scale comparably to libc.

References

[1] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices 35*, 11 (2000), 117–128.