CSC469 Assignment 2 Report

Jessica Hypatia Boritz, Wing Yan Grace Li, Yuchen Zeng

Section 1: Introduction

Allocation and management of memory is key in any modern system, and in the study of system designs, it is crucial to understand the design decisions made to keep the allocator up to par in four areas of performance: speed, scalability, false sharing avoidance, and low fragmentation. This report describes our version of a parallel memory allocator, discussing its design, implementation details, and its performance.

Section 2: Design

The design of our allocator was based upon `kheap` and Hoard. The general idea behind our allocator is the same as in `kheap`, being a pool-based subpage allocator, but we have modified it to better scale with a larger number of processes via the use of fine-grained locking, and to avoid false sharing. As with `kheap`, pages of free memory are tracked by linked lists of `pageref`.

Because large allocations (> ½ pagesize) were explicitly stated to be rare, the methods for handling memory allocations and frees have been split into two cases: the small case, for allocations of ½ pagesize or less, and the large case, for allocations of more than half a page.

Large Allocations

The majority of the code for handling large allocations was reused from the provided `kheap` code, and was serialized via the use of a lock, `big_list_lock`. Note that this is only serial with regard to other large allocations: small allocations use their own data structures and locks, and given that large allocations themselves are rare, it should be extremely unlikely to have multiple concurrent large allocations, so this serializing should not lead to a major performance drop. For large allocations, we allocate 8 bytes more memory than needed to store metadata. We store -1 in the first 4 bytes of the first page, indicating that it is a large allocation. The metadata allows better management of different types of allocations.

Per-processor Heap and Block Class Size

Adapted from Hoard, for small allocations, we maintain a per-processor heap for each processor, which contains B `pageref` for each supported block class size, where B is the total number of block class sizes. The per-processor heaps are managed as a continuous chunk of memory, each containing the head of linked lists for page references. Management involves two indices: the heap index and the block size index. Assuming that each thread executing requests is exclusive to a single CPU, the heap index is calculated by taking the CPU ID. The block size index is sorted according to the size of the block class, starting from 0.

Since we are aligning pages to 8 bytes, we decided that the block class size should be a power of 2, with the smallest supported block class size being 8 bytes. The subsequent sizes are obtained by doubling the previous size, up until we reach half of the page size (for our implementation on x64_64, this is 2048 bytes). Starting from 8, this approach yields 9 lists of page references for each per-processor heap. For example, assuming CPU ID are starting from 0 and increment by 1 in order, the linked list of blocks with class 16 bytes own by cpu 3 will have heap index 2 and block size index 1, and be the 2*9 + 1 = 19th page reference linked list.

By using size classes that are powers of 2 apart and rounding the requested size up to the nearest size class, we limit the worst-case *internal fragmentation* within a block to a factor of 2. To reduce *external fragmentation*, we recycle completely empty pages for reuse by any size class, and by any thread.

Metadata

One unique aspect of our design is the practice of associating metadata with each page. When a call is made to `mm_malloc()`, and if we allocate a new page to manage this memory, we store two integers of additional metadata in the first 8 bytes of space. The first is the processor index, calculated by CPU ID mod number of processor, this identifies the per-processor heap that this page of memory belongs to. The second piece of metadata is a “block class size”. Similar to `kheap`, we keep a list of supported sizes, in which this “block class size index” indexes into the list of supported sizes that we keep.

We ensure that memory is allocated in multiples of the page size, allowing us to locate the beginning of a page given any pointer within that page in constant time. By storing the metadata at the beginning of a page, we can access it quickly when `mm_free()` is called, and use them to locate the per-processor heap that is to be accessed for freeing this memory.

Small Allocations

When a small allocation is received, we determine the heap index and the block size index for the allocation, and use this to access the list of page references that these indices map to. We iterate through the `pageref`, looking for the first page with free space. If we cannot find one, we instead allocate a new `pageref` from a shared list of free `pagerefs` and add it to our list. If we can not obtain a new `pageref` from the lists of free or recycled references, we allocate a page’s worth of `pagerefs`, ensuring that each `pageref` is on a distinct cache line to avoid active false sharing. Once we have a page with free space, we take a chunk of memory from its freelist, write our metadata to this memory, and return a pointer to the desired memory.

Whenever a new page is allocated, the first 8 bytes of space are used for metadata. As a result, there is a special case where the first block of the page can store at most the size of block class minus 8 bytes of memory. In this case, the address returned to the user is 8 bytes beyond the start of the allocated region. The metadata stored is the heap index and the block size index. This allows us to find the page that an allocation belongs to and free it in constant time,

ensuring that freed memory is returned to the heap it came from, thereby avoiding passive false sharing.

When a small allocation is to be freed, we obtain the metadata stored with the allocation, and use it to access the list of page references our allocation came from, iterating through the list until we find the page it came from. We then restore this memory to the page reference, and check to see if the entire page of memory associated with this `pageref` is unused. If this is the case, we remove the `pageref` from the list, and add it to a list of reusable `pagerefs`.

Access to the list of free `pagerefs` is serialized using a lock. Additionally, all calls to `sbrk` to allocate more memory are serialized with another lock. Access to each processor's heap (its per-size lists of `pagerefs`) is also serialized, using a per-heap lock. Because we know that a processor will only have a single thread executing on it at once, this lock may seem redundant. However, because freeing memory does not always occur on the same processor, multiple threads may attempt to access the same processor's heap simultaneously, necessitating this lock.

As discussed above, similar to Hoard, we do not free pages of memory obtained from the system back to the system virtual memory. Instead, we place these empty pages to a freelist, such that the allocator could reuse these now-empty pages. These empty, reusable pages are reused in a LIFO order, such that it is more likely to immediately use a page that is already in cache. These mechanisms that are in place, helps bounding blow-up, avoiding the allocator constantly obtaining memory from `mem_sbrk` endlessly.

Section 3: Design Motivation and Alternatives

In designing our allocator, the motivating goal was to avoid false sharing (both passive and active), and contention on shared data structures by creating per-processor data structures that only a single thread would act on at once. If this was the case, locking would not be needed. However, our desire to always return freed memory to the heap it came from meant that it was possible for multiple processors to access the same heap. Additionally, to allow freed memory from one processor to be used on another processor, we have to allow for a shared data structure to point towards currently unused memory (in our case, the lists of new and reusable free refs). Thus, although we have mostly independent data structures, we needed to add locking for all of our data structure. This allows for a higher chance for congestion to occur when multiple threads attempt to access these shared data-structures, in turn for bounding the blow-up and being able to reuse claimed memory from the OS, and it is a tradeoff that we are willing to make.

In the process of implementing and finalizing our allocator, we first implement a base code that is focused on correctness, while keeping in mind the goal of our implementation: avoiding false sharing. After base code is confirmed to be correct, our main goal is to optimize based on analysis of weaknesses of the base code. We noticed rather high fragmentation. Our allocator uses more memory in total, compared to other allocators. This is partially due to the fact that we

only return fully-empty pages to the global list of free refs, and also the fact that in the base code, we associate 2 integers of metadata to each piece of allocated memory, instead of associating it with the page of memory, creating unnecessary redundancy that leads to high memory usage. The modification to store this metadata at the front of each page of memory helped address this issue.

In an attempt to lower fragmentation further, an attempt was made to adopt the use of empty threshold, as described in Hoard. By adding the amount of memory in use in the heap u_i, and the amount of memory allocated by our allocator to the heap from the system a_i, we are able to calculate the empty threshold f. Once the page of memory hits the emptiness threshold, we return the pageref to our global free ref list, and allow for it to be reused. This would have been helpful since we have noticed that some threads would acquire a page, but have it be mostly empty with little memory allocated. Since our algorithm only returns a page when it is entirely empty, the thread holds on to the mostly empty page without doing anything with it. With the emptiness threshold adopted, in theory, it should help avoid low usage of pages, thus lowering fragmentation. There would be a tradeoff between lowering fragmentation, and potentially promoting false sharing with this approach, but it is worth it since fragmentation is too high. However, this idea was not adopted in our final implementation, mainly due to time constraints. The attempt for implementing this is recorded and can be accessed under the branch `emptiness_threshold`.

Another attempt to lower fragmentation in parallel, is to put pageref objects at the front of the page, such that the metadata that describes the page is stored within the page itself. This, in theory, should lower fragmentation since there is not the need to allocate a page's worth of memory for pageref objects. Plus, the fact that we allocate pagerefs in increments of cache line size to avoid false sharing, and the size of a pageref just so happen to be ½ of cache line size, meaning that we could have made better use of that empty space. Placing pageref at the front of the page keeps the property of them residing on different cache lines to avoid false sharing, while providing opportunity to better use of allocated memory, lessen the amount of empty, unused, wasted space. It would also allow for freed big_malloc pages to be reused for small_malloc allocation, since the change in structure would mean that each and every page has the same structure, and it is easier to prepare the pages for reuse. However, this is also not adopted due to the same reason as the last idea, and is available under the branch `pageref_at_pagehead`.

Section 4: Results

Testing our design and the allocator's performance, we ran benchmark programs on multiprocessors. These benchmarking programs also allow for us to demonstrate its speed, scalability, false sharing avoidance, and fragmentation.

The platform used, which is a dedicated server provided for benchmarking our allocator on MarkUs, has 20 Intel(R) Xeon(R) Gold 6209U CPU installed, at 2.1 GHz, with cache-size of 28160 KB, total RAM 196480028 kB. For testing, the programs will only use up to 8 cores

though the system has access to 20. The system runs Linux 5.15.0, and all programs were compiled with GCC 11.0, with c99 standard at the -O3. The following benchmarking programs were used: *cache-scratch*, *cache-thrash*, *linux-scalability*, *threadtest*, *larson*, and *phong*. The specifics and differences of each program will be discussed along with the results presented below.

For simplicity, we will call our allocator `a2alloc` for the rest of this report.

Sequential Speed

As stated in the Hoard paper, a good parallel allocator that is scalable and memory-efficient, should perform serial operations fast enough that it can be comparable to a sequential allocator, such that it guarantees performance good performance when dealing with single-threaded programs, or when a multithreaded program is run on a single processor.

	libc	**a2alloc**	**Increase in Speed(%)**
cache-scratch	1.15216	1.152846667	0.06%
cache-thrash	1.152430333	1.147634333	-0.42%
linux-scalability	0.01726966667	0.9963343333	5669.27%
threadtest	0.5682086667	0.635091	11.77%
***larson* (throughput)**	24405005	4536347.667	-81.41%
phong	0.8455596667	0.8545633333	1.06%

Table 4.1: Sequential speeds of libc and a2alloc, running on different benchmarking programs

We ran all benchmarking programs single-threaded, and obtained their runtime (or throughput, for larson) for each. Compared to `libc`, which is designed to be a parallel allocator to start with, the performance of `a2alloc` is similar to that of `libc` in the benchmarks *cache-scratch* and *cache-thrash*, where they test for false sharing. This is expected as our design is specially targeted to avoid false sharing at all cost. In most benchmarks, `a2alloc` obtained results that are comparable to that of `libc`. However, it performs exceptionally badly in *linux-scalability*. Given that the `libc` allocator is a state-of-the-art allocator that is commonly used these days, the change in speed is maintained within +/- 15% is impressive. A lower than optimal sequential speed is also a tradeoff that was made to have better scalability, less false sharing, while keeping the allocator relatively performant in terms of sequential speed.

Scalability

4 of the 6 benchmarks are designed such that scalability of the allocators can be observed. We ran these benchmarks across the allocators that we would compare against: `libc` and `kheap`.

The *larson* benchmark, different from others, simulates a server. Each thread allocates, deallocates and also deliberately transfers some objects for other threads to free. This is not

tested in other benchmarks, and is significant since this shows how well an allocator deals with workloads distributed across threads, and the throughput of such tests would vary greatly depending on the design decisions made for each allocator.

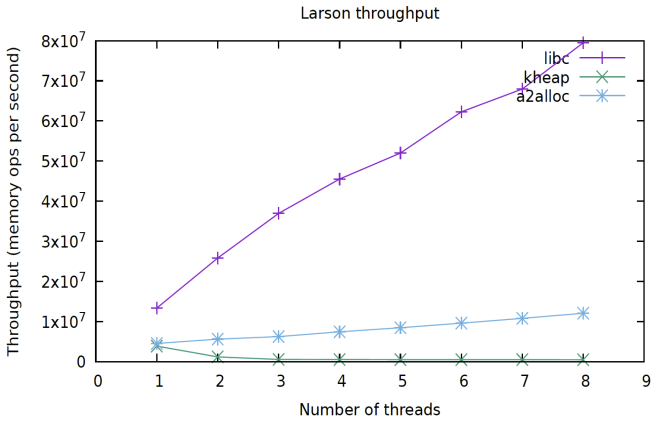

Figure 4.1.1: Throughput against number of threads, running Larson benchmark

Figure 4.1.2: Speedup as number of threads grows, running phong benchmark.

It is clear from Figure 4.1.1 that the `libc` implementation outperforms both `a2alloc` and `kheap`, while a2alloc was able to outperform `kheap`. At 8 threads, `a2alloc` was able to obtain roughly 26 times the throughput of `kheap`. The speedup of `a2alloc` is only about half of the performance of `libc`, as shown in Figure 4.1.2.

Phong, adapted from the Hoard paper, randomly selects the allocation sizes, and the timing of freeing the objects allocated. As such, the benchmark pushes the allocators in terms of freeing objects that are definitely not sequentially located in memory.

Figure 4.2: Speedup as number of threads grows, running phong benchmark.

Here, both `libc` and `a2alloc` speed up exponentially, with a2alloc being at a slightly slower rate. Until 3 threads, the scale-up is similar and slowly drifts apart.

FIgure 4.3: Speedup as number of threads grows, running on *threadtest* benchmark

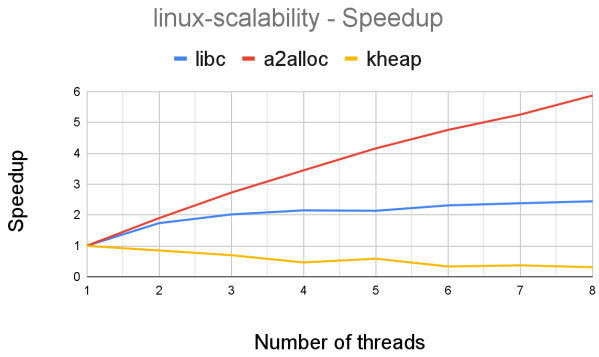

Figure 4.4: Speedup as number of threads grows, running on *linux-scalability* benchmark

Threadtest and *linux-scalability* are two basic scalability benchmarks. With *threadtest*, the threads allocate a number of objects, do some computation, then free these objects. It repeats this with different numbers of objects, though, the total number of objects allocated for each run is approximately the same, regardless of how many threads are being used.

Linux-scalability, on the other hand, does not do work on the allocated objects, and each thread would allocate the same amount of objects, such that the total workload grows linearly as the number of threads grows. As shown in Figure 4.4, for *linux-scalability*, `a2alloc` outperforms all other allocators, while `libc` would not speed up much after it reaches 3 threads, and `kheap` has worse performance as the number of threads grows.

On the contrary, in *threadtest*, where it has work done after allocating the objects, `libc` outperforms the two, while `a2alloc` no longer scales up as it reaches 4 threads. It still outperforms `kheap`, where `kheap` does not scale well, showing that it is not optimized for parallel workloads.

This difference in performance between the two benchmarks can be explained by the increased possibility of congestion when performing the *linux-scalability* tests. Since there is no work to be done in between allocations, depending on implementation, it can increase the chances of congestion.

False sharing

There are two types of false sharing, passive and active. Active false sharing occurs when multiple threads intentionally share variables that are close together in memory, updating them concurrently. While passive false sharing occurs when threads are accessing different variables, that just happen to be located closely in memory. To investigate whether the allocators avoid false sharing, *cache-scratch* tests for passive, and *cache-thrash* tests for active false sharing. Since `a2alloc` is designed intentionally to avoid false sharing, it is no surprise that it performs well compared to other allocators.

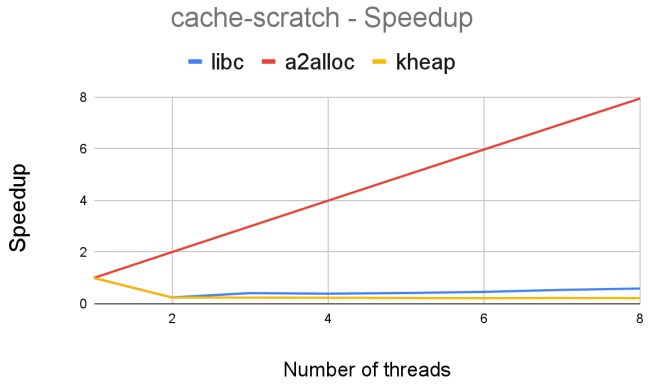

Figure 4.5: Speed up as number of threads grows, running on *cache-scratch*

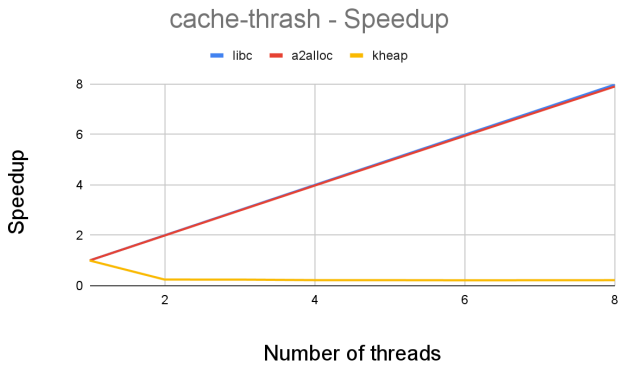

Figure 4.6: Speed up as number of threads grows, running on *cache-thrash*

For *cache-scratch*, it outperforms all other allocators with flying colors, scaling up linearly, while other allocators do not handle cache scratching well.
This is thanks to the different design decisions for avoiding passive false sharing. We have deliberately avoided putting any variables that would be shared between threads on the same cache line. Shared data structures, such as per-processor mutexes, pageref objects, are all put into different cache lines at initialization, such that threads would not have to access the same cache line to access these commonly accessed variables and objects.

On the other hand, for cache-thrash that measures active false sharing, `libc` and `a2alloc` have almost identical performance, where `kheap` still does not scale well. This is expected as the design of `a2alloc` is aimed at avoiding false-sharing. The superb performance in these benchmarks indicates our success in avoiding false sharing.

Fragmentation

Fragmentation, as defined in the Hoard paper, is the maximum amount of memory obtained from the OS, divided by the maximum of memory needed by the application (i.e. the actual total size of memory that has to be allocated to applications by our allocators). As stated in the Hoard paper, it is difficult to calculate and measure the actual fragmentation from our test set-up. Thus, we would be using the maximum size that allocators need to grow the heap to complete each benchmark, for the analysis of fragmentation. This is also the “Memory Used” output of the benchmark programs used.

Single-threaded Applications

We compared the fragmentation of kheap and a2alloc, by extracting the memory usage tracked by the benchmarking programs. Results are compiled in Table 4.2. Observing the table of data, it is quite clear that the difference in memory usage is significant.

	kheap (bytes)	a2alloc (bytes)	**Increase in Memory Usage**
cache-scratch	12287	20479	66.67%
cache-thrash	12287	16383	33.34%
larson	372735	704511	89.01%
linux-scalability	13709311	14233599	3.82%
phong	1990655	1998847	0.41%
threadtest	12287	20479	66.67%

Table 4.2: Comparing memory usage of kheap and a2alloc, running on 1 thread

In *linux-scalability* and *phong*, the memory usage of a2alloc is similar to that of kheap, indicating fragmentation is less significant. However, all other benchmarks indicate that a2alloc uses much more memory than it is needed from the application. This can be due to a2alloc needing to allocate pages of metadata, including pageref objects, at the initialization stage. Another possible reason is the fixed sizes that a2alloc supports, and how we allocate those sizes. Due to the goal to always free memory to where it came from, we pre-slice all memory slices into suitable sizes, while keeping metadata at the front of a page for fast freeing. This, in turn, causes the page of memory to not be fully used, when the page of memory cannot be sliced further after containing metadata. This indicates room for improvement to reduce unused space in the per-processor heaps, as well as for structures that are shared.

Multithreaded Applications

Similar to single-threaded applications, the allocators are run with multiple threads, up to 8 threads, for each benchmark. We then extract the memory usage tracked by the benchmarking program, and compare the usage of a2alloc with that of kheap, to estimate fragmentation. Since the benchmarking programs that have random workload are scaled such that the total workload of all threads are the same no matter the number of worker threads used, the average is not skewed.

	kheap (bytes)	a2alloc (bytes)	**Increase in Memory Usage**
cache-scratch	14847	33791	127.59%
cache-thrash	12287	30719	150.01%
larson	1592148.333	3074388.333	93.10%
linux-scalability	44025513.67	63955796.33	45.27%
phong	3107156.333	3287209.667	5.79%
threadtest	17407	49151	182.36%

Table 4.3: Comparing memory usage of kheap and a2alloc, averaging the memory usage of runs on 1 to 8 threads

It is obvious that a2alloc uses much more memory than that of kheap, except for *phong*, indicating that a2alloc handles random workload well, with a random timeline for freeing objects. It also does better in *linux-scalability* than in the rest of the benchmarks, indicating that

fix-sized memory allocation is easier on `a2alloc`. The same that is said for single-threaded programs can be reiterated for multi-threaded programs, and that high fragmentation is a weak point of `a2alloc`.

All in all, there is significant room for improvement in fragmentation for `a2alloc`.

Section 5: Conclusion

Our aim for an allocator that avoids false sharing, has succeeded. Even compared to state-of-the-art allocators, it is comparable in terms of avoiding false sharing. Scalability is fair, not as good as `libc`, but comparable. Though, our decision to avoid false sharing at all costs, and to speed-up the malloc and free sequential speeds, holds us back on fragmentation.

In the case that further development is to be done on `a2alloc`, the first issue to tackle would be internal fragmentation. A good starting point would be to evaluate the failed attempts for tackling this exact problem, as described in section 3. Next would be to improve on sequential speed.

The development of such a parallel allocator, has shed light onto the design decisions of existing allocators, and its limitations, and has furthered our understanding in the study of system design. A small change in a seemingly unrelated component can be catastrophic.

Section 6: Bibliography

Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: a scalable memory allocator for multithreaded applications. *In Proceedings of the ninth international conference on Architectural support for programming languages and operating systems (ASPLOS IX).* Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/378993.379232