Assignment 2 Report

Koko Nanahji, Ali AhmadiTeshnizi (Grad)

November 16, 2020

Contents

1 Introduction

Memory allocation is a vital functionality needed by almost any program, and therefore performance and efficiency of the memory allocator plays an important role in the performance of the whole system. Furthermore, the increasing number of multi-threaded applications necessitated the development of concurrency aware memory allocators. Hoard allocator [1] is one of the many works addressing the problems in regard to this matter. In the paper, authors suggest that an acceptable memory allocator must have the following characteristics:

1. Speed
2. Scalability
3. False Sharing Avoidance
4. Low Fragmentation

In this work, we have implemented and evaluated multiple methods for handling memory allocation. In the first section, we will explain the details of different approaches we have considered and tried. In section two, we will present the performance evaluation results. Finally, we will compare these methods from various aspects and come to an integrated conclusion.

2 The Allocators

We used k-heap and libc memory allocators as baseline implementations, and tried to present a comparison with those in our approaches. We assumed that the size of the actual pages in the system is 4096 bytes (which is referred to as *pagesize* throughout this document since it can be easily modified for different sizes) and cache line size is 64 bytes and considered three allocator architectures:

2.1 Simple Linked-List-Based Allocator

This allocator consists of a set of heaps: one global heap and one heap per processor. An overview of the structure of a single heap in this allocator is available in figure 1. Each process "owns" the heap with ID corresponding to the ID of the processor it is currently using. Since it is stated that there will be at most one thread executing in a single CPU at any given time, this mapping of threads to heaps seemed the most appropriate. Furthermore, this specification also made spinlocks more suitable compared to other (heavier) locks because we know that the threads will not be contending for CPU time and can spin while waiting for the lock without interfering with the performance of the other threads.

In this architecture, for allocations that are at most 2048 bytes we use pages that are divided into equal sized blocks, where the sizes of the blocks are powers of 2 starting from 8 bytes until 2048 bytes and for larger allocations we use as many contiguous pages as needed. Each heap has a list of pages that are completely free, a list of pages which are

completely full (that contains pages with block sizes at most 2048), for each block size a list of pages that are partially full (i.e. each page has at least one free block and one used block) stored in an array called *sizebases* (we also refer to these lists as partially full lists), and a list for large allocations (i.e. with blocks larger than 2048 bytes). Furthermore, it also contains a list of previously allocated page reference objects that are currently unused and a spinlock per list for all the lists to handle concurrent accesses.

We refer to each page by a *pageref* that "owns" that page. Therefore, when we allocate a new page we need to get a new *pageref* that holds metadata regarding that page such as the list of free blocks in that page, number of free blocks in that page, block size of that page etc. When allocating a new page, we first check the list of free *pagerefs* and use one of them if there are any. Otherwise, we allocate a new page, divide it into a list of *pagerefs* and add them to the free *pagerefs* list to be used later.

Initially, all of the lists are empty. The $mm\_init()$ function initializes the heap. When the user wants to allocate x bytes by calling $mm\_malloc(x)$, we first identify the block size that needs to be allocated. If $x > \frac{pagesize}{2}$, we use $large\_malloc()$ and allocate $\frac{blocksize+pagesize-1}{pagesize}$ pages for the block. Note that this might lead to allocating extra blocks (up to one minus number of bytes in a page i.e. 4095 bytes), however, the fragmentation caused by $large\_malloc()$ is negligible since the number of large allocation (greater that $\frac{pagesize}{2}$) are assumed to be rare.

On the other hand, if the requested size is at most $\frac{pagesize}{2}$, then we use $small\_malloc()$ function to allocate block of size $max(8, 2^{\lceil log_2(x) \rceil})$ which proceeds as follows. It first looks at the first page in the partially full list that contains the list of pages with block size $max(8, 2^{\lceil log_2(x) \rceil})$. If there is an empty block there, it removes that block from the free list of blocks of the corresponding page and returns a pointer to the address of that block. If that list is empty (i.e. no free blocks), it looks at the free pages list. If there is a page there, it takes that page sets the block size of that page to be $max(8, 2^{\lceil log_2(x) \rceil})$ and removes one of the blocks. Furthermore, it adds this page to head of the partially full list of the corresponding block size in the *sizebases* array. If the free pages list is also empty, it checks the global heap's list for free pages and if there is a page there then it removes that page, sets the block size of that page to be $max(8, 2^{\lceil log_2(x) \rceil})$ and removes a block from that page. Then, it adds this page to the partially full list of the corresponding block size in the *sizebases* array of it own heap. Finally, if there are no free pages available in global heap's free list, it allocates a new page, sets the block size of that page to be $max(8, 2^{\lceil log_2(x) \rceil})$ and removes a block from that page and adds it to the appropriate entry in the *sizebases* array of its own heap.

When the user wants to free a block pointed by *ptr* by calling $mm\_free(ptr)$, given the internal structure of the memory allocator, we need to figure out the page in which the block pointed by *ptr* resides and free as only the appropriate block. We first identify the *pageref* that owns the page in which the block, pointed by *ptr*, resides. This is done by a linear search over partially full, complete, and large pages lists in all the heaps (starting from the heap it owns). If the pointer is for an allocation that is at most 2048 bytes, the appropriate block is removed from the corresponding page. In case where this block was the only used block in that page, we move that page to the free pages list of the heap in which that block originally resided. If the page is in a complete pages list, we add the block to the free list of that page and move the to the appropriate entry in the *sizebases* array (based on the size of the blocks in the page) in the heap that it resides.

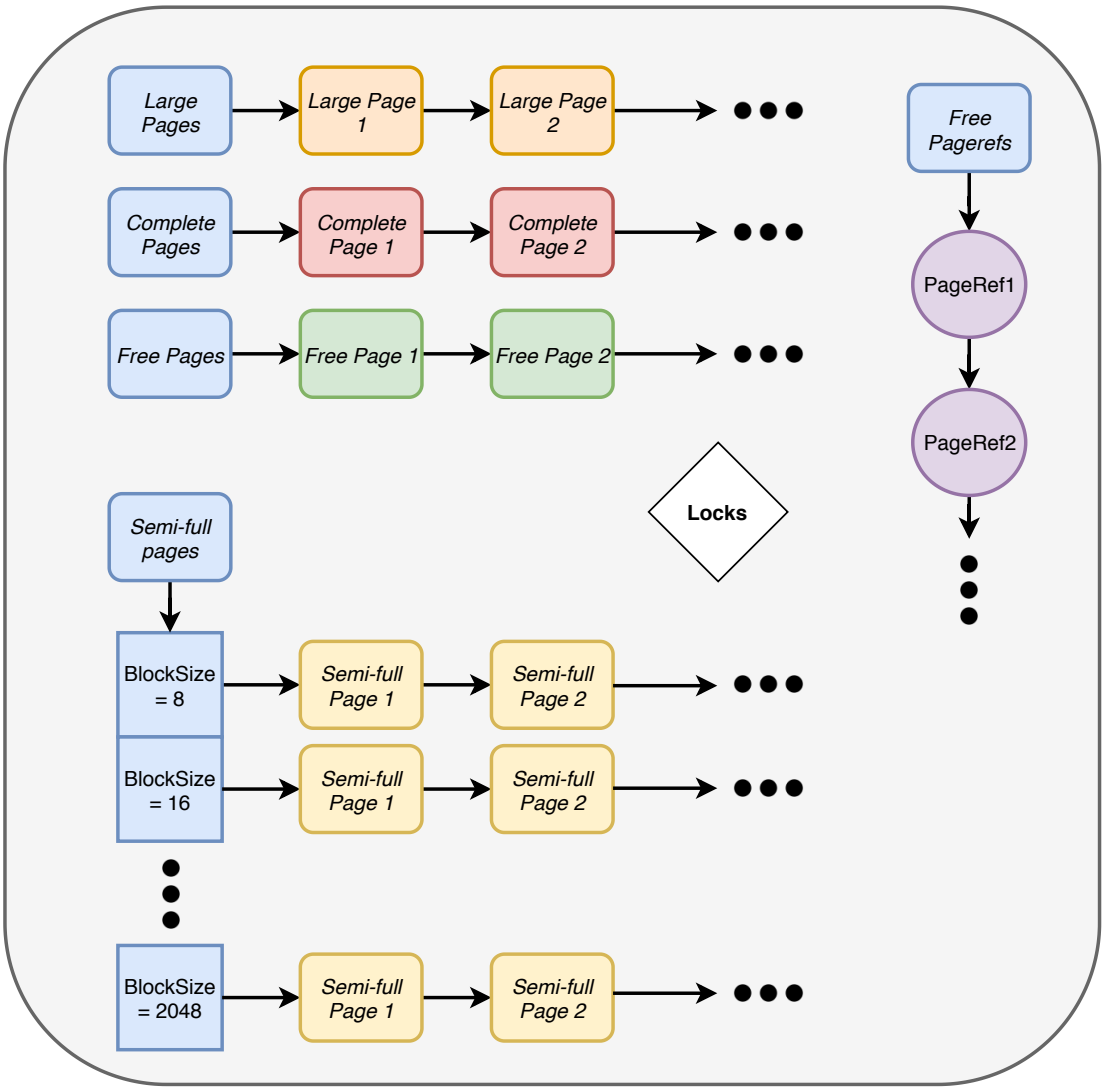

Figure 1: Simple Linked-List-Based Allocator Heap Structure

If the free pointer corresponds to a large block, then the free function divides the pages allocated by that block into *pagesize* pages and adds them to the free pages list of the heap in which the block belonged. Each time we add a new page to the free list we check to see if the number of free pages in the heap is larger than 2 (this number was chosen experimentally), if so we move a page to global heap's free pages list. This strategy of moving only completely free pages to global heap helps eliminate passive false sharing.

The worst-case time complexity for allocating a new block is $\mathcal{O}(1)$ (once the appropriate locks have been acquired), however, the worst-case time complexity of a free operation is $\mathcal{O}(n)$ where n is total number of pages in all the heaps that contain at least one non-free block (excluding the time spent spinning on the locks). The performance analysis that we conducted showed that the linear search in the pages used by the allocator is a performance bottleneck.

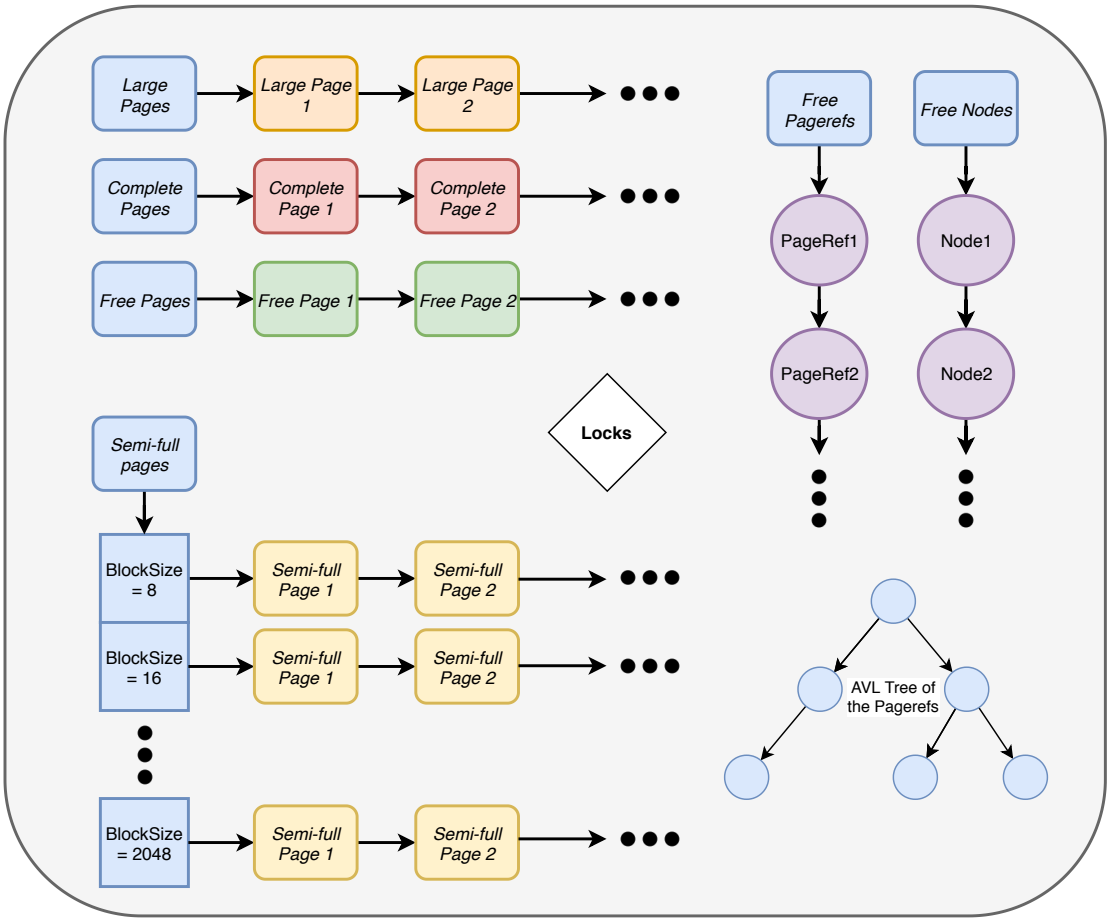

Figure 2: AVL-aided Linked-List-Based Allocator Heap Structure

2.2 AVL-aided Linked-List-Based Allocator

Based on the analysis of previous approach we improve the performance of the allocator by adding an AVL tree [2] to each heap to keep track of the pages that contain at least one non-free block. This tree mapped the addresses of the pages to their corresponding *pageref* objects which allowed the allocator to avoid the search over many pages to find the appropriate page during a free operation. With the addition of the AVL trees the worst-case time complexity for free became $\mathcal{O}(\sum_{h \in H} log(n_h))$ where H is the set of all heaps and n_h is the number of pages that contain at least one non-free block in heap h (excluding the time spent spinning on the locks). However, the worst-case time complexity of insert in heap h became $\mathcal{O}(log(n_h))$ (in including the time spent spinning on the locks). To keep track of the nodes in the tree we create a list of free nodes which is used in parallel with free *pagerefs* list. The new design is available in figure 2. We implemented the AVL tree from scratch, as there are no appropriate libraries providing the desired functionality in C. Although, the performance of this approach was better than the previous one, the performance of the allocator was worse than the libc's memory allocator which led us to try another approach which will be discussed in the next section. We are including the code for the tree in the *final_report* directory of the repository, named as *avl_alloc.c* for reference.

2.3 Linked-List-Based Allocator with Internal *Pagerefs*

Our final approach is to store the metadata of each page (i.e. the pageref) within the actual page, more specifically for each page the *pageref* of that page is located in the first couple of bytes of that page. If we align the pages allocated by the allocator with the *pagesize*, we can easily identify the address of the page in which a given block resides (by taking the modulo of the address with pagesize). Therefore, we can directly find the appropriate *pageref* objects from the given address of a block and the worst-case time complexity for both insertion and deletion becomes $\mathcal{O}(1)$ (once the appropriate locks have been acquired). However, this might result in high amounts of fragmentation, especially for larger block sizes, as a portion of each page is set aside for the *pageref* objects. We mitigated the problem by doubling the pagesize used in our pages (i.e. the page sizes used in the allocator became twice of the page size in the system which is $2 \cdot 4096$ bytes) and we kept everything else the same (i.e. block size ranges for pages in *sizebases* array stayed from 8 bytes to 2048 bytes). Furthermore, to each *pageref* object we added a previous pointer to make the time it takes for removal of a page from a list constant and added an integer in each *pageref* object that indicates the ID of the heap in which the *pageref* belongs. We tried to make the page size larger than twice of the actual page size of the system, however, our experiments showed no improvements in the performance of the allocator. The overall structure of the allocator is the same as the figure 1 with the free *pagerefs* list removed.

3 Experimental Results

We executed all the benchmarks for all three allocators on *csc469-2* machine which has 20 processing cores and page size 4096 bytes and was relatively free at the time of the experimental runs (we did not include more details on this system since it has widely used in this course). The results are presented in the following subsections. Considering all the performance charts, internal *pageref* design shows a superiority comparing with the two other designs. Considering the memory usage of these allocators, we can see that the structural complexity of our allocators is resulting in a higher memory fragmentation. Libc has almost no fragmentation in some tests (Linux-scalability and phong) and has a relatively small memory overhead in the other benchmarks. Kheap has a considerable more fragmentation, but it is mostly less than our allocators. As we expected, the internal allocator and the AVL allocator have higher memory overheads comparing with the simple allocator due to their designs.

3.1 Threadtest

Threadtest benchmark tests the basic scalability of the allocator. Each thread repeatedly allocates number of objects, does a fixed amount of computation, and then deletes its objects. The number of allocations per thread is scaled so that the total number of allocations is roughly the same regardless of the number of threads used. Here, the simple linked-list-based allocator (called "the simple allocator" hereafter) performs almost as good as libc, even though the search for freeing is naive (since each thread removes objects from its own heap). Both the AVL-aided linked-list-based allocator (called "the AVL

allocator") and the linked-list-based allocator with internal *pagerefs* (called "the internal allocator" hereafter) show a similar performance due to the structure of the benchmark (as it does not stress the free operations much).

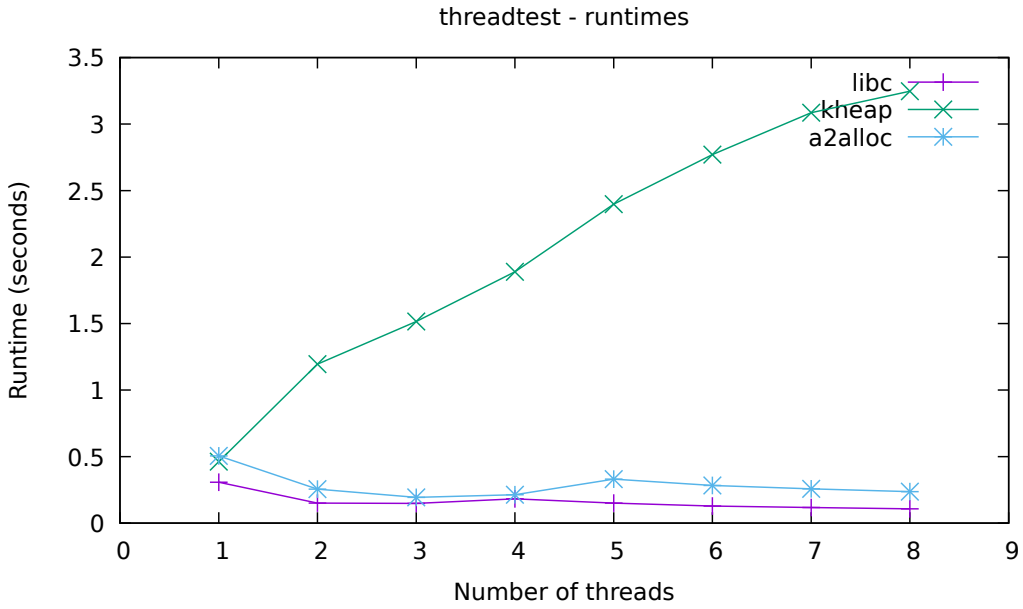

Figure 3: Threadtest results for Simple Linked-List-Based Allocator

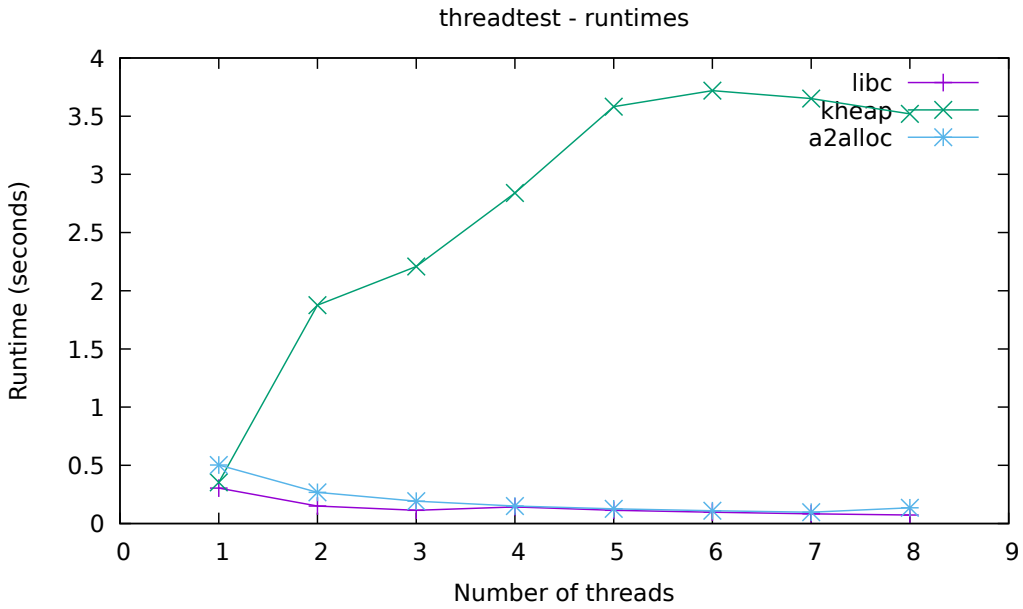

Figure 4: Threadtest results for AVL Linked-List-Based Allocator

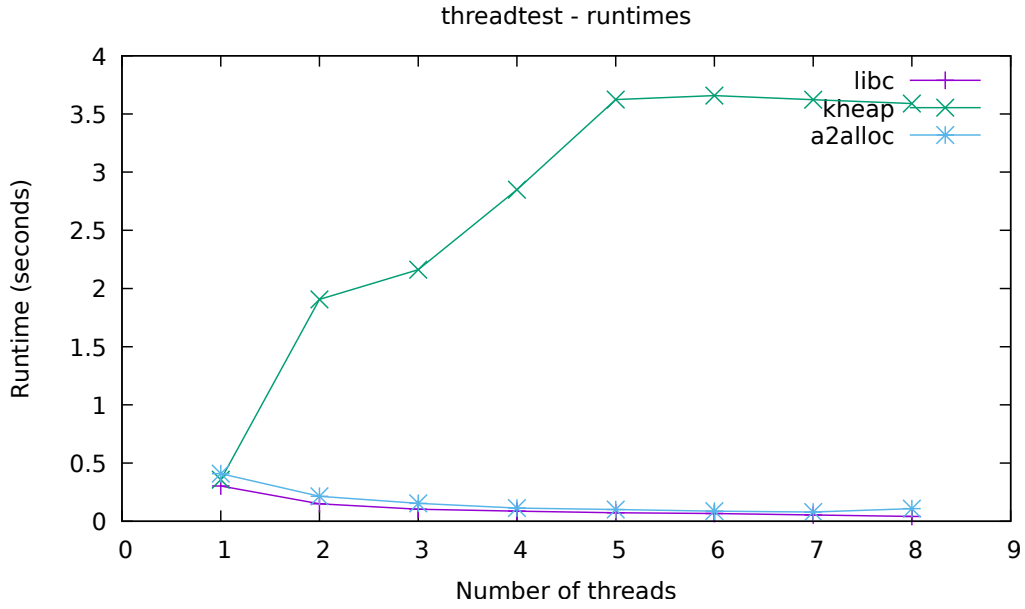

Figure 5: Threadtest results for Internal Linked-List-Based Allocator

3.2 Cache-Scratch

All of the methods outperform both libc and kheap in Cache-scratch benchmark. This is because our allocators all have lazy mechanisms to move pages to global heap, and they can handle this benchmark effortlessly.

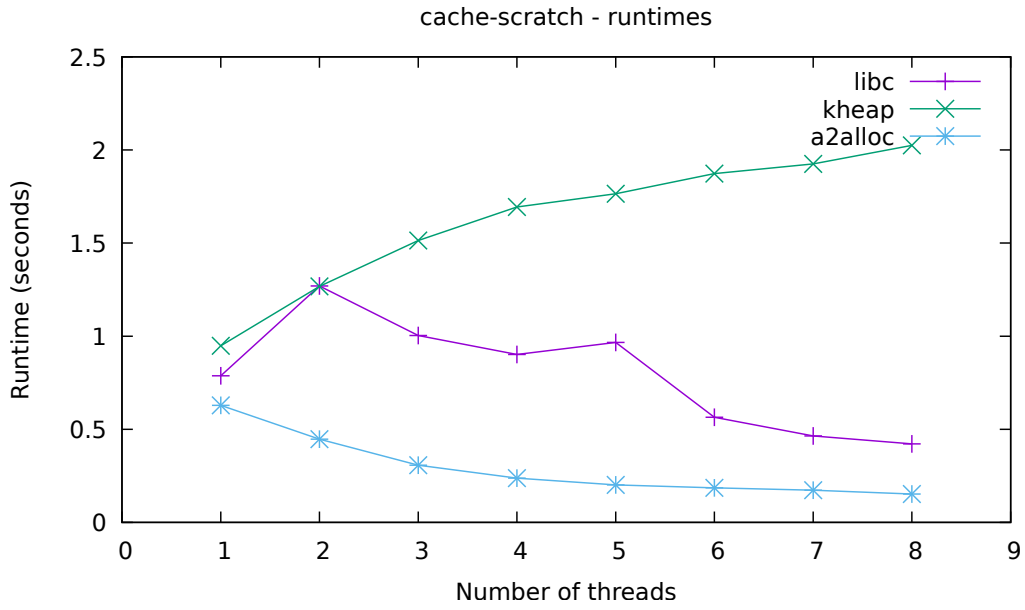

Figure 6: Cache-scratch results for Simple Linked-List-Based Allocator

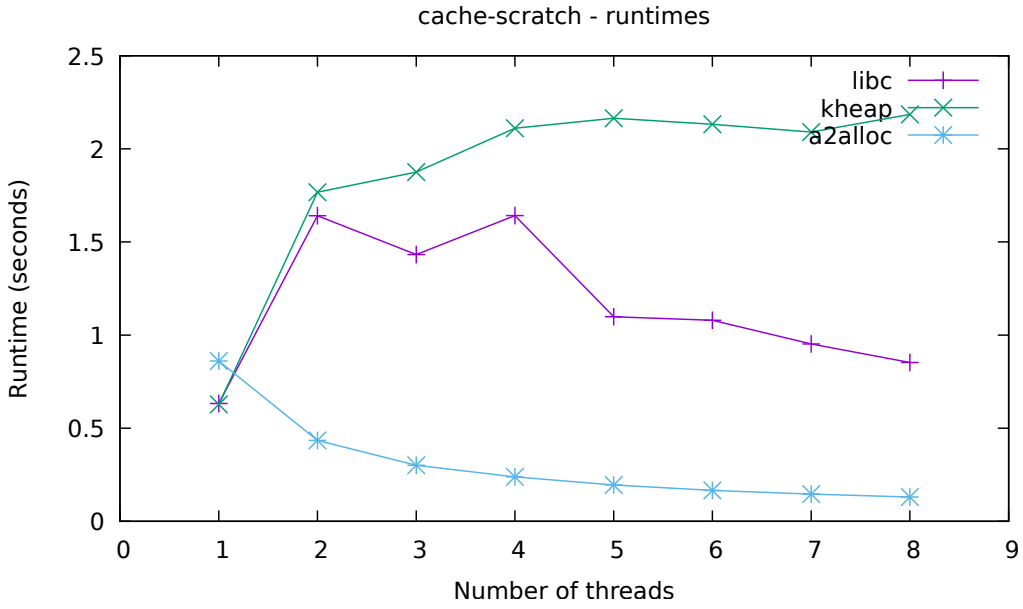

Figure 7: Cache-scratch results for AVL Linked-List-Based Allocator

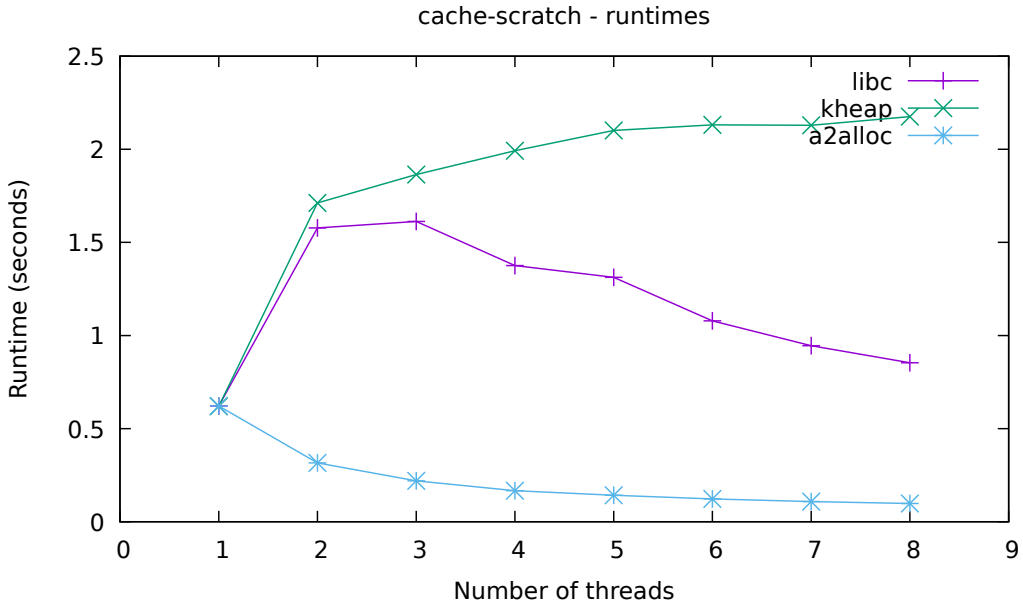

Figure 8: Cache-scratch results for Internal Linked-List-Based Allocator

3.3 Cache-Thrash

Cache-thrash benchmark focuses on the active false sharing, and as our allocators are using separate heaps for threads executing on different processors, they perform in the same ballpark as libc and better than kheap.

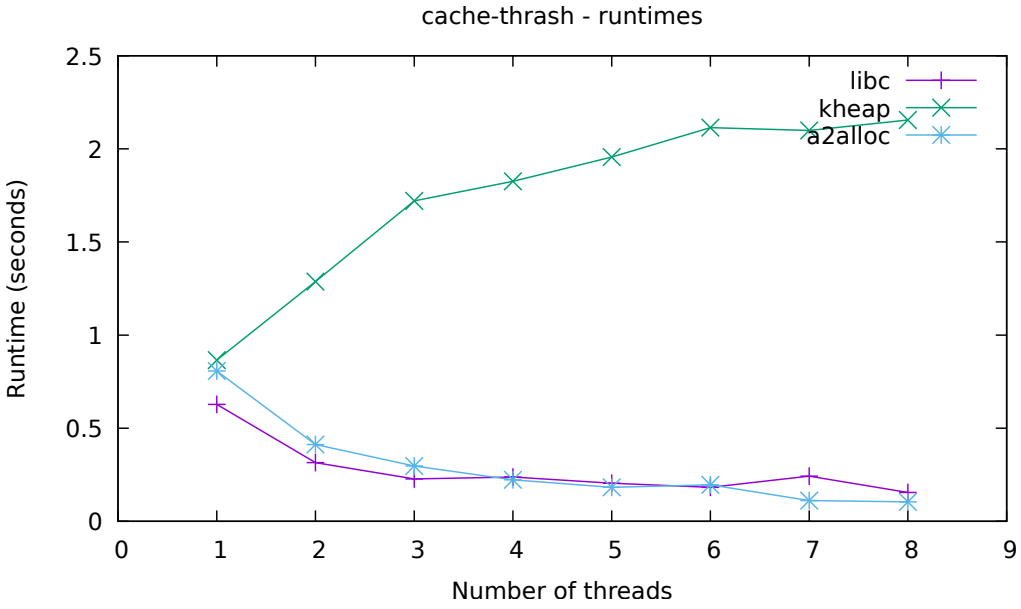

Figure 9: Cache-thrash results for Simple Linked-List-Based Allocator

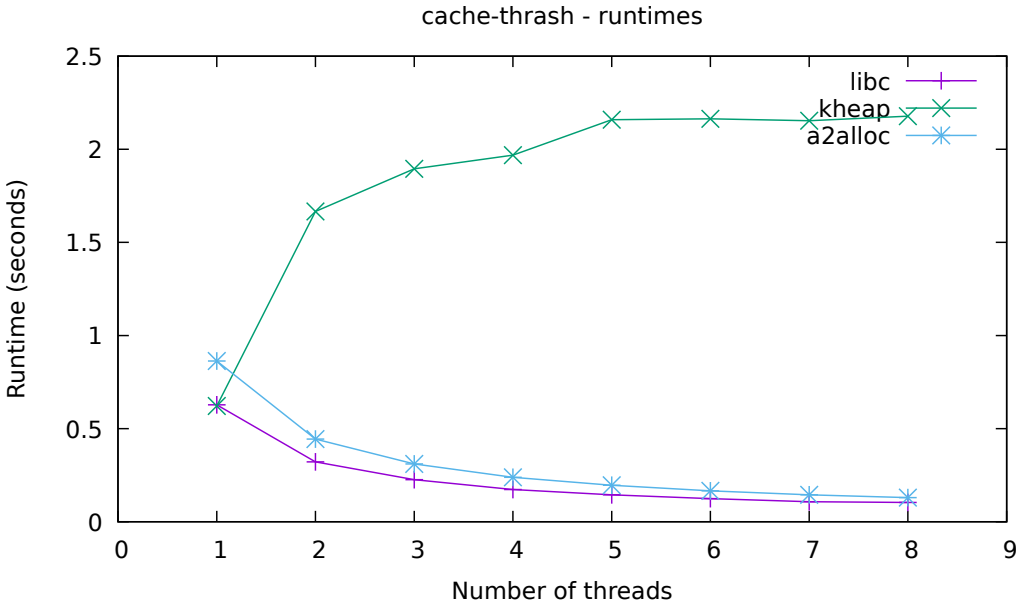

Figure 10: Cache-thrash results for AVL Linked-List-Based Allocator

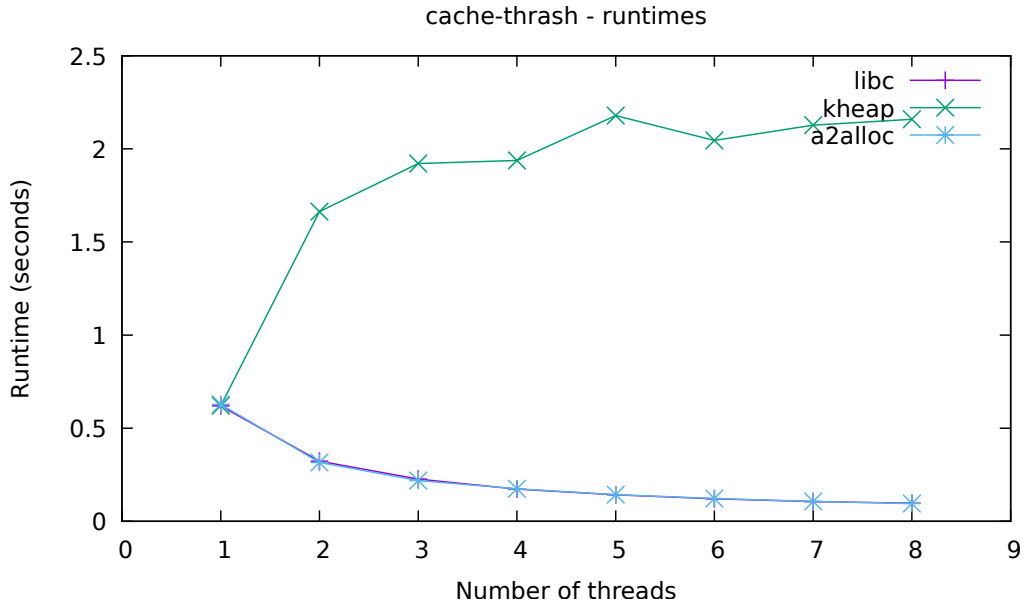

Figure 11: Cache-thrash results for Internal Linked-List-Based Allocator

3.4 Phong

This benchmark does a random selection of allocation sizes, and randomly chooses an allocated item to free. Our allocators are moderately better than libc on Phong benchmark. The superiority increases for the internal allocator because of the faster free operations.

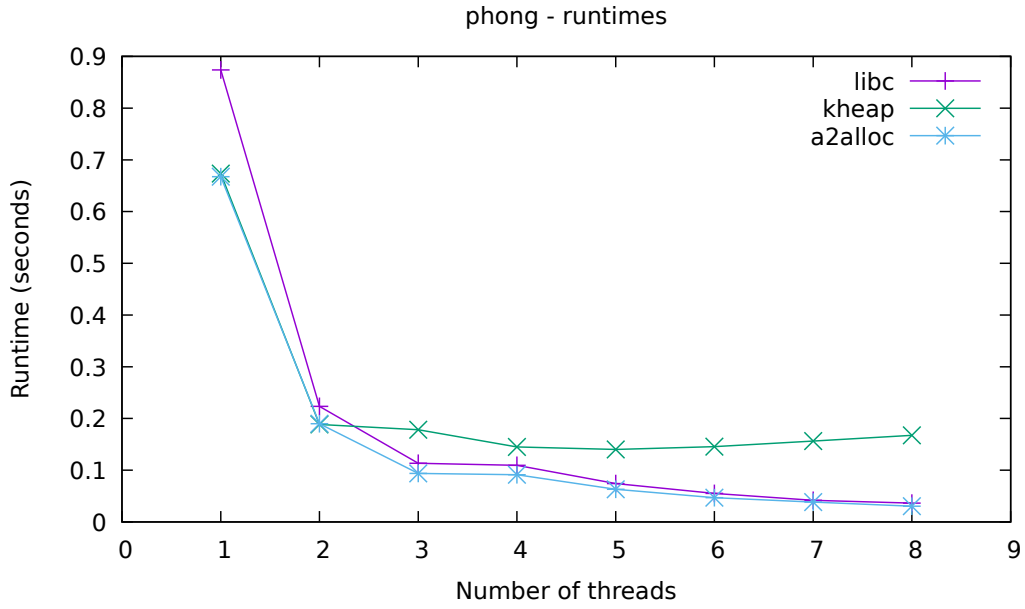

Figure 12: Phong results for Simple Linked-List-Based Allocator

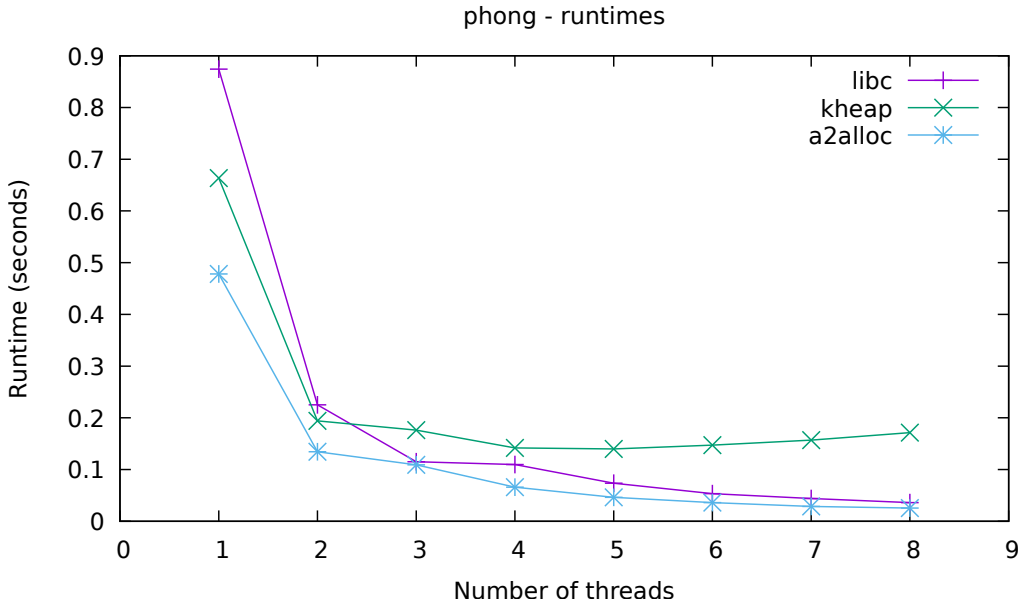

Figure 13: Phong results for AVL Linked-List-Based Allocator

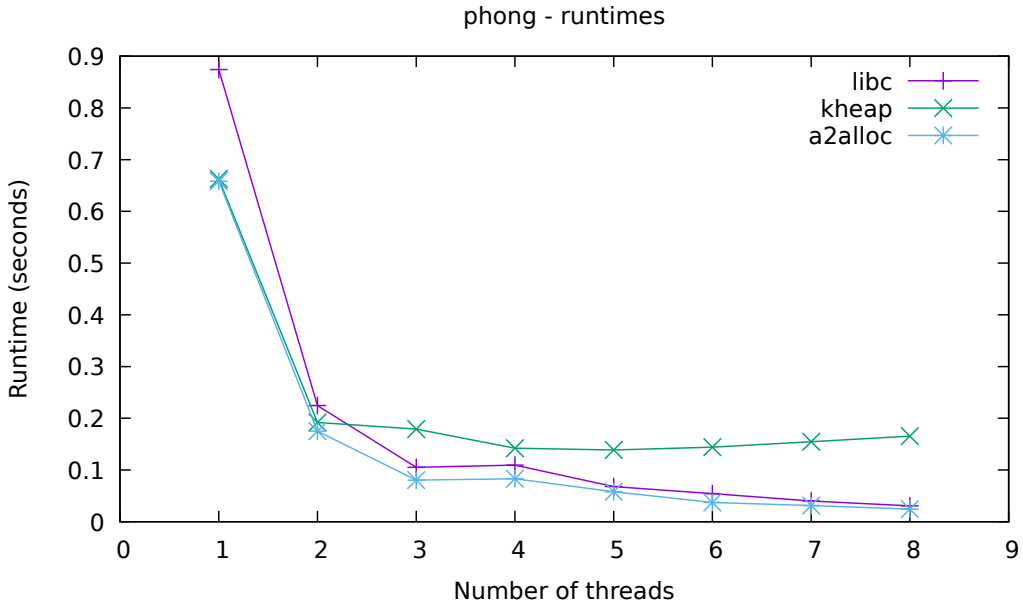

Figure 14: Phong results for Internal Linked-List-Based Allocator

3.5 Linux-Scalability

Considering Linux-scalability benchmark, we can clearly see that unlike kheap, our allocators smoothly scale up with the number of threads. The extra insertion and deletion overhead of the AVL heap is causing it to be slower than the other two allocators.

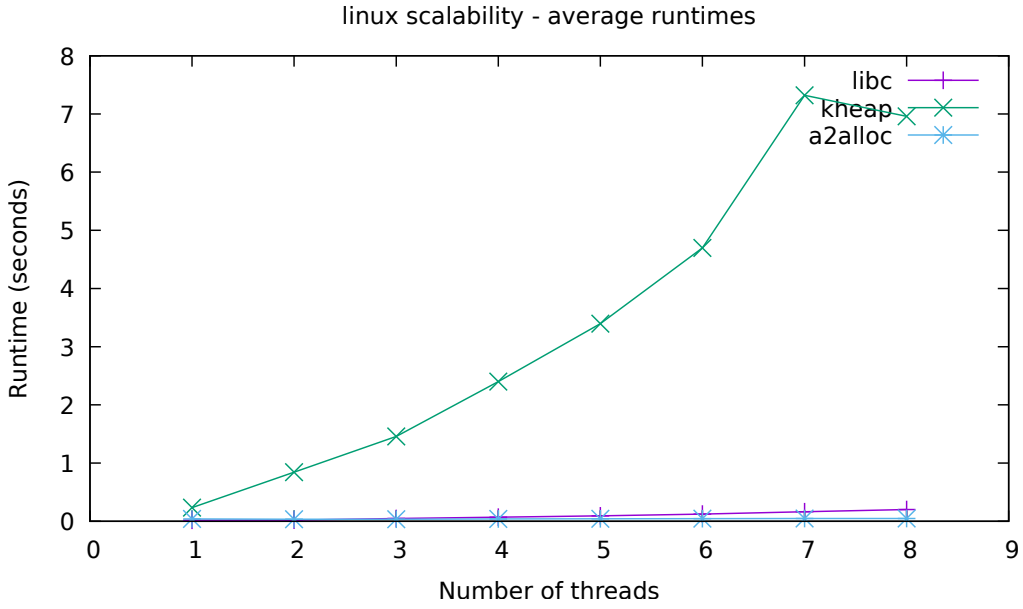

Figure 15: Linux-scalability results for Simple Linked-List-Based Allocator

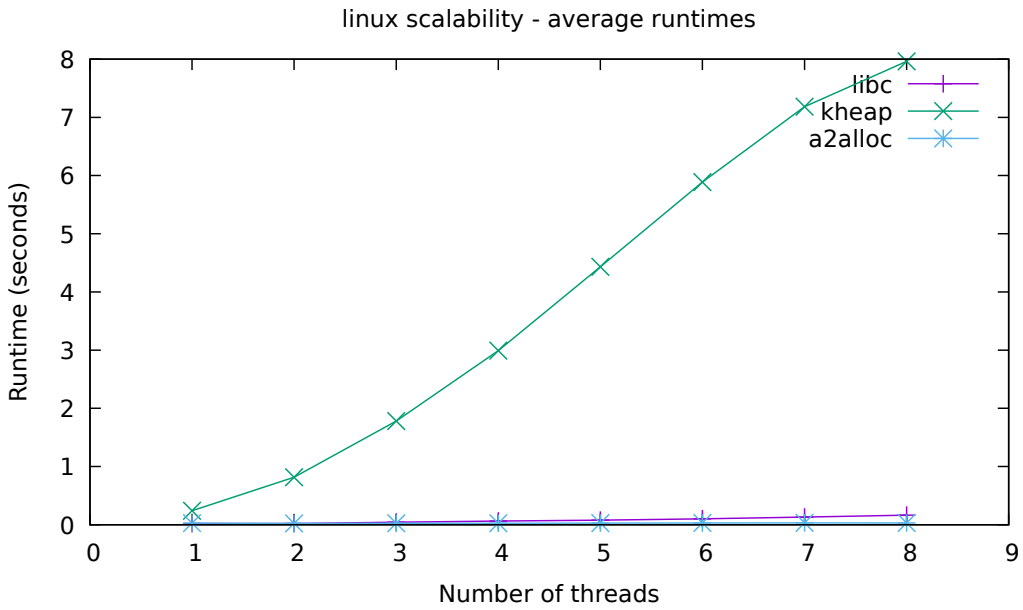

Figure 16: Linux-scalability results for AVL Linked-List-Based Allocator

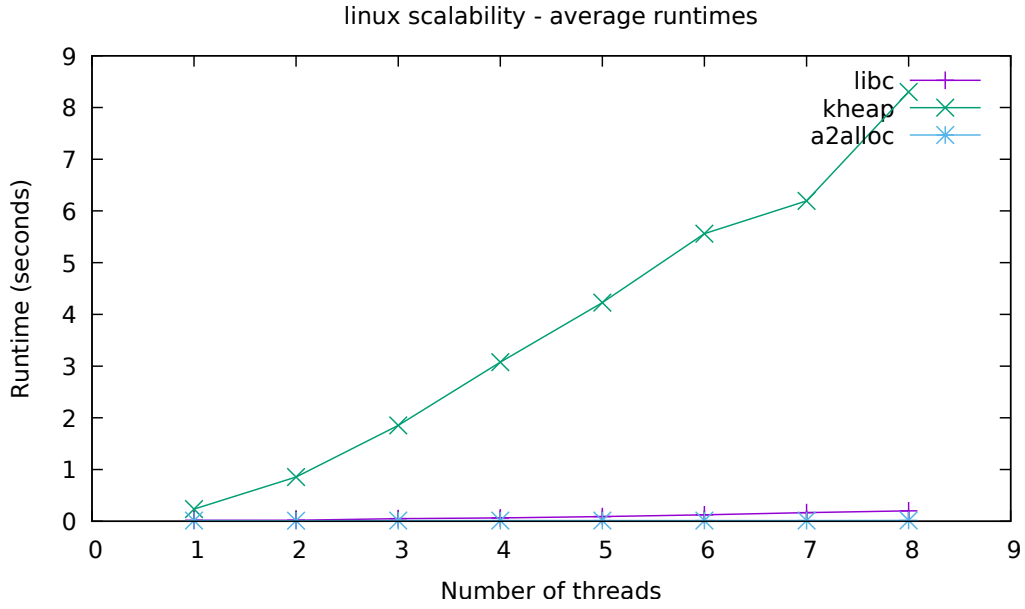

Figure 17: Linux-scalability results for Internal Linked-List-Based Allocator

3.6 Larson

Finally, Larson benchmark tests the throughput over highly parallel operations. There was a problem with the AVL allocator causing a deadlock in this case, so the results for that plot are not reliable. However, Larson benchmark plots for the two other allocators (which are reliable) show that they perform well in parallel workloads. We can also see that the simple allocator is not as good as the internal allocator, especially for smaller number of threads. This is because of the large number of free operations executed by threads on different cores than the cores used by the threads that allocated those blocks.

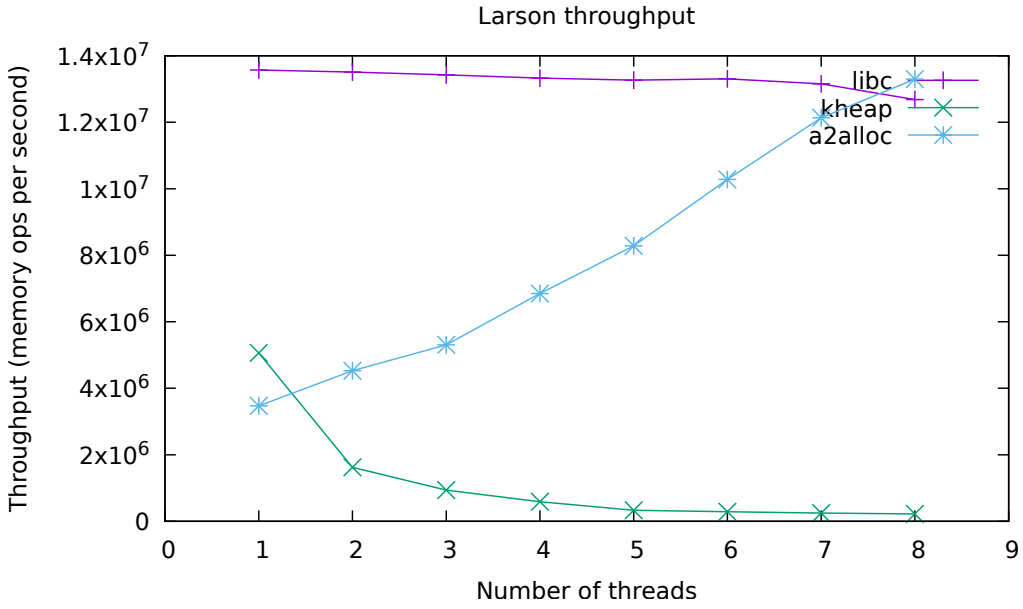

Figure 18: Larson results for Simple Linked-List-Based Allocator

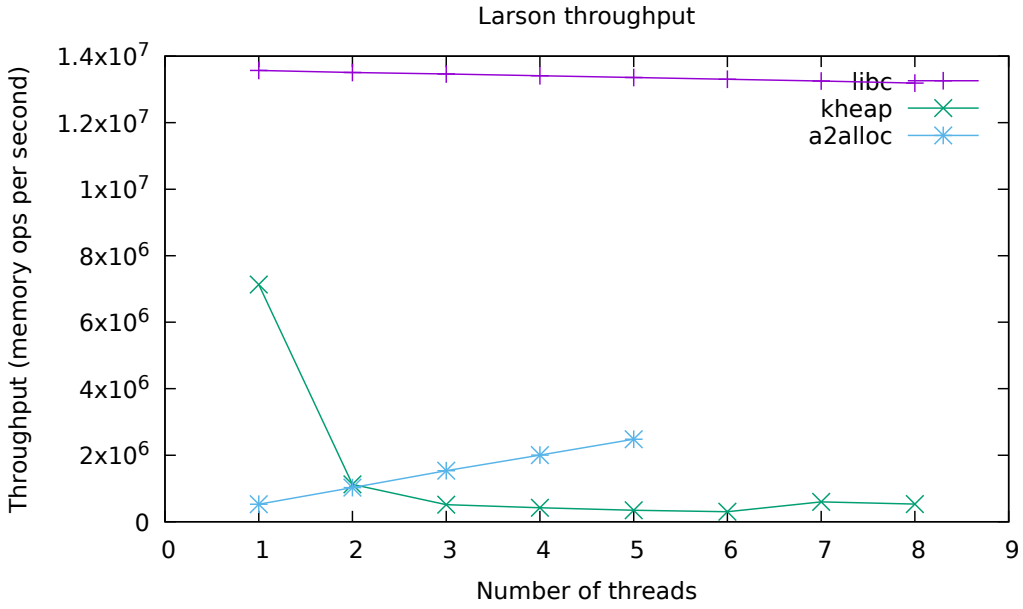

Figure 19: Larson results for AVL Linked-List-Based Allocator

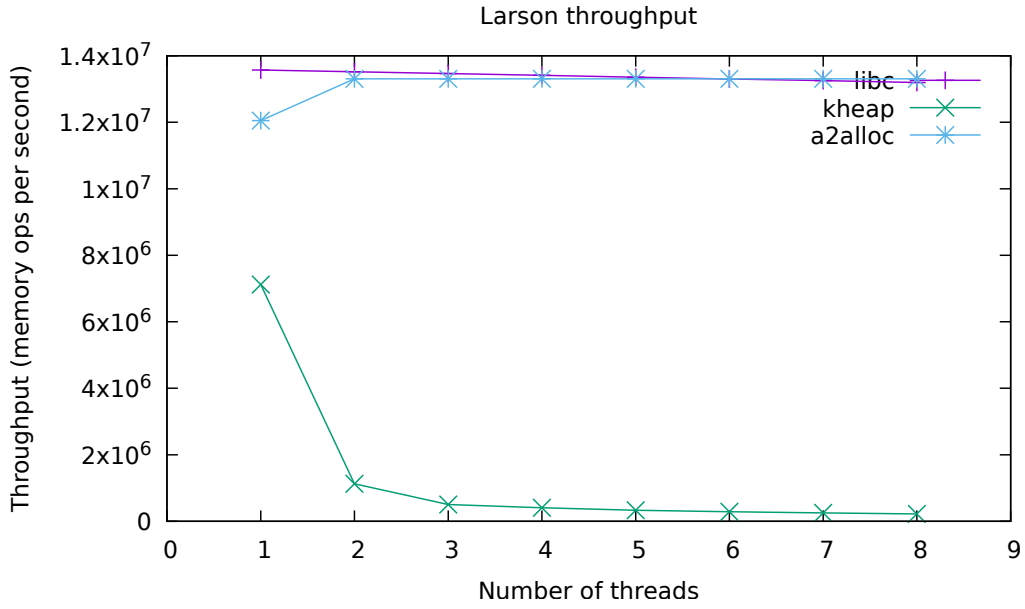

Figure 20: Larson results for Internal Linked-List-Based Allocator

4 Conclusion

In conclusion, we can see that simple modifications allow us to design allocators that are even better than widely-used ones. The best allocator we implemented was the allocator that stored *pagerefs* within the corresponding pages, as the worst-case complexity for different operations in this allocator were significantly low. Our experimental results also confirmed this. That allocator is as fast as serial allocator, scalable, with no passive false sharing and in most use cases creates moderate memory fragmentation.

References

[1] Emery D. Berger et al. "Hoard: A Scalable Memory Allocator for Multithreaded Applications". In: *SIGARCH Comput. Archit. News* 28.5 (Nov. 2000), pp. 117–128. ISSN: 0163-5964. DOI: 10.1145/378995.379232. URL: https://doi.org/10.1145/378995.379232.

[2] Robert Sedgewick and Kevin Wayne. *Algorithms*. 4th. Addison-Wesley Professional, 2011. ISBN: 032157351X.