CSC469 Assignment 3
Parallel Memory Allocator

Authors

Jasmeet Brar - 1002342967
Jasmeet Sidhu - 1002469571

November 13, 2019

Contents

1 Implementation **1**

2 Results and Analysis **2**

List of Figures **I**

List of Tables **II**

References **III**

1 Implementation

Based on the Hoard paper, the allocator was constructed with the concept that every processor would have their own heap, and there would be a global heap that is shared among all the processors [1].

The very first time that `mem_init` is called, 256 MB + 2 pages of memory is allocated to the program, and the first page is reserved to store global data structures, such as each processor heap's data structures. The processor heap structures simply contain the lock for synchronization, a pointer to the linked list of all superblocks owned by the heap, and a pointer to free pages. Superblocks are the size of two pages (8 KB on most systems), and are used for allocations less than or equal to 4 KB. This idea is also taken from the Hoard allocator since it reduces false sharing. All allocations which are greater than 4 KB are stored directly in contiguous pages. For example, a 6 KB allocation would be placed in two 4 KB pages.

Several algorithms were used to limit blowup and limit fragmentation by reusing memory blocks as much as possible. All small allocations were rounded to the nearest power of 2 within the range 32 to 4096 bytes. Rounding allocations to such sizes helped in maintaining linked lists of different block sizes in the superblocks. Then when an allocation comes in, the heap can simply search for an appropriate block size in the appropriate list. Furthermore, the lists of free blocks are sorted during runtime to make coalescing and merging blocks efficient. If a block is freed and the list finds a block of the same size, the 2 blocks can be merged into one block; similarly, if the heap needs a smaller block but it only sees bigger blocks, it can split the large blocks into smaller blocks. This strategy improves internal fragmentation by reusing memory as much as possible.

Similarly, when a large allocation is freed, the heap uses linked list structures to keep track of the starting points of free contiguous pages. Then when more pages are needed, the heap simply searches through the list for pages which can be reused.

Like Hoard, this allocator keeps pages and superblocks inside the processor heap for as long as possible. This behaviour makes the allocator more scalable because reusing pages and free memory blocks reduces contention on the global heap which can only be accessed serially. Hoard's

2 Results and Analysis

The memory allocator has been benchmarked and tested by the provided test servers. From the results, we have found that our memory allocator has performed considerably in both Cache Scratch and Cache Thrash tests for false sharing, where its run time would be less than 2 seconds, as shown in Figure 1 and Figure 2. This was the result of having every processor have its own heap, which is page aligned, to be where it would do its allocations, and as explained in the Hoard paper, a superblock that is being released by one processor tends to be completely empty, so we would be able to avoid false sharing there, and this event doesn't occur very often [1]. Hence we were able to do very well in the two tests, and our performance comes on par with libc.

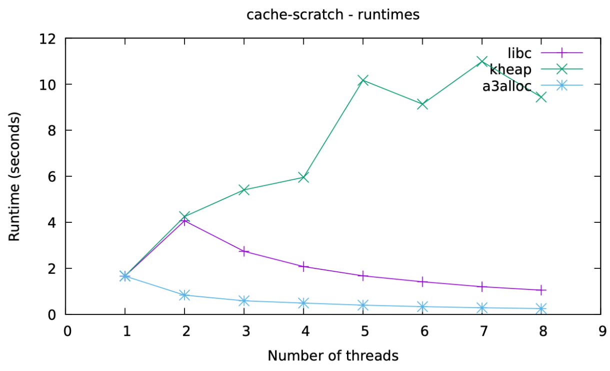

Figure 1: Cache Scratch Test

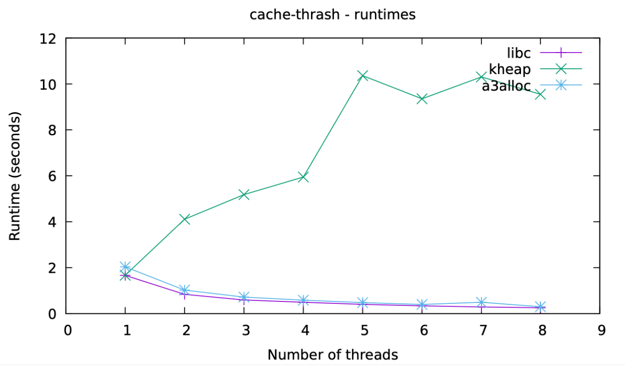

Figure 2: Cache Thrash Test

As for the Larson Throughput test, we notice that the serial implementation's throughput stays around the same as we increase the number of threads. This is due to the fact that only one thread can allocate or free memory at a time. As for our implementation, since we are allowing more threads to access the memory concurrently, the throughput would naturally increase. It wouldn't be at the level of libc where the throughput tends to stay around constant as the number of threads increase, and this is due to varying factors, such as that our allocator has to maintain data structures such as linked lists and that takes a toll on the performance.

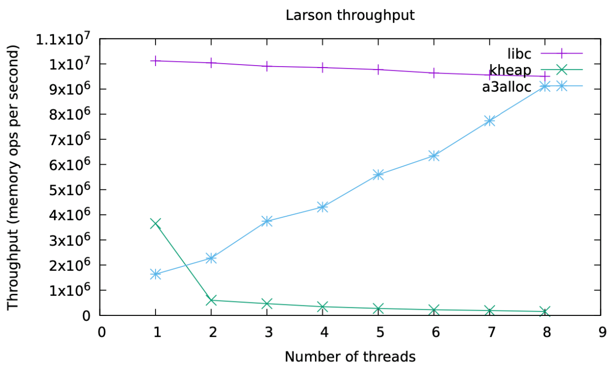

Figure 3: Larson Throughput Test

As for Linux Scalability test, our allocator doesn't seem to perform as well as libc and kheap, and this may be due to a tradeoff between performance and reduced fragmentation. Our allocator took time to attempt to find the best fit for the memory that needs to be allocated, and also update pointers for all the memory operations in order to reduce fragmentation. Considering that for this test, the workload get evenly distributed for all threads, there's bound to be contention and that is an example that one would see in Figure 4.

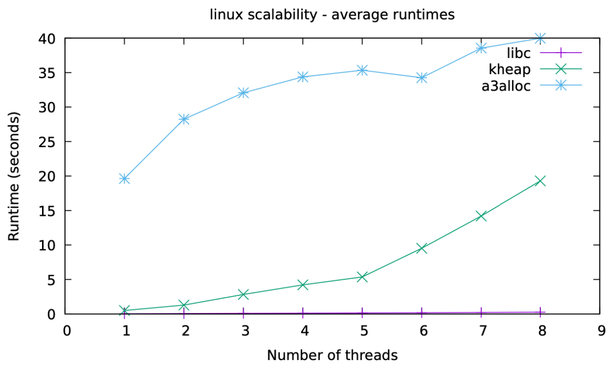

Figure 4: Linux Scalability Test

As for the Phong test, because of the random allocations that it was making that is unusual, all three allocators would on average take the same hit on performance, which means that they would all perform about the same. However, kheap is a serial implementation, so it will take an additional performance hit.

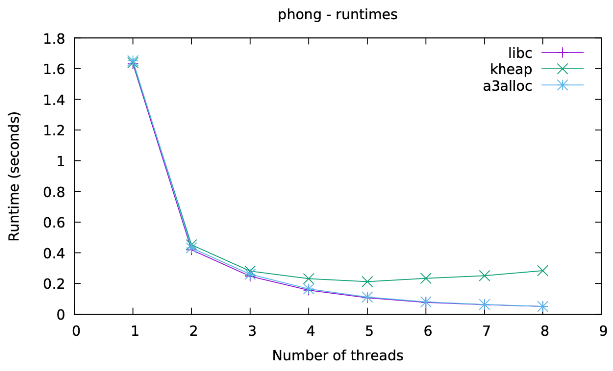

Figure 5: Phong Test

As for the thread test, our allocator performed similarly to libc, so we have decent scalability there.

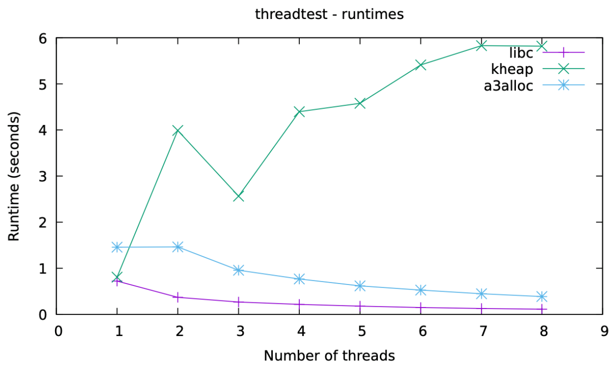

Figure 6: Thread Test

One alternative algorithm we tried was to distribute the processor heaps between threads was to use $2 * num\_processor$ heaps, like Hoard, and to mod into them based off of the thread ID. This solution was found to have slightly lower performance than the current implementation of using the physical core number, so it was not used. Of course this was only possible due to the given assumption that each thread only runs on one physical core, which is not always the case in real systems.

List of Figures

1 Cache Scratch Test 2
2 Cache Thrash Test 2
3 Larson Throughput Test 3
4 Linux Scalability Test 3
5 Phong Test 4
6 Thread Test 4

List of Tables

References

[1] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, “Hoard: A scalable memory allocator for multithreaded applications,” *SIGOPS Oper. Syst. Rev.*, vol. 34, pp. 117–128, Nov. 2000.