Multicore Processors: Architecture & Programming Project

Project Group No.: 2
Project Title: Memory Allocator for OpenMP Programs

Project Group Members:

1. **Name**: Darshan Dinesh Kumar
   NetID: dd3888
   University ID: N10942768

Abstract

The end of Moore's Law necessitates the development of innovative solutions to augment the performance of applications rather than attempting to pack more transistors on a chip and/or increasing the CPU frequency. In this direction, Multicore and Multiprocessor systems are now ubiquitous and are an ideal candidate to improve performance as they enable the execution of parallel, multithreaded programs. However, these parallel applications are often inhibited by the memory allocator which can negatively throttle performance and scalability. Moreover, the memory allocator can also introduce other issues such as false sharing and fragmentation which can be considerable bottlenecks to performance. This project introduces a scalable memory allocator that can be used in conjunction with these parallel applications (even those developed with OpenMP). It introduces the concept of per thread heaps with memory ownership that can scale efficiently while almost eliminating false sharing and minimizing fragmentation. The generated results for the developed OpenMP benchmark programs denote substantial promise with the proposed and implemented memory allocator exhibiting considerable improvements in Scalability, False Sharing avoidance and low Fragmentation as compared to the Malloc and Hoard memory allocators.

Introduction

Moore's Law which stated that the number of transistors on a microchip doubles every 18 months—has slowed down significantly. For decades, Moore's Law drove the advancement of computing power, allowing for exponential gains in performance simply by making transistors smaller and increasing their number. However, as physical and economic constraints make it increasingly difficult to further shrink transistors, the emphasis in computing has shifted.

Rather than relying on traditional methods of increasing clock speeds or transistor density, innovative solutions are now required to augment performance. These solutions are no longer just about raw hardware improvements; they also need to focus on optimizing software systems to fully leverage the available hardware. This has led to a marked shift towards parallel computing—a paradigm that enables multiple tasks to be executed

simultaneously. The transition from single-core to multicore and multiprocessor systems has been at the forefront of this shift. These systems enable the simultaneous execution of multiple threads or processes, allowing programs to handle more computations in parallel, which can result in significant performance improvements, especially for computationally intensive applications.

Multicore and multiprocessor systems, which consist of multiple processor cores or multiple processors working in tandem, have become ubiquitous in modern computing, from smartphones and laptops to large-scale server farms and high-performance computing clusters. These systems are ideal candidates for improving performance, as they allow for greater parallelism. In theory, this parallelism should allow applications to scale better with the number of cores, enabling faster execution times for multithreaded programs. Programs that can be divided into smaller tasks that run concurrently as parallel threads can potentially achieve linear or near-linear speedup with the addition of more cores, resulting in massive improvements in processing speed and efficiency.

However, the effective use of multicore and multiprocessor systems is not as straightforward as simply adding more cores. While the hardware can support parallel execution, the software—specifically, the memory allocator—often becomes a bottleneck in achieving optimal performance. A memory allocator is responsible for managing the allocation and deallocation of memory during program execution. In multicore environments, memory allocators can throttle performance and scalability for several reasons.

In a multithreaded environment, multiple threads may attempt to allocate and free memory simultaneously, leading to the need for synchronization mechanisms such as locks. If these locks are not efficiently managed, they can result in significant contention and serialization of memory operations, which ultimately reduces the parallelism that the system can exploit.

Another important challenge related to memory allocation in parallel systems is false sharing. False sharing occurs when multiple threads access different variables that are located on the same cache line. When multiple threads modify variables that are located on the same cache line, the CPU cache becomes inefficient because each thread's modification forces the cache line to be invalidated and reloaded from memory (in other words, the cache coherency protocol kicks in), resulting in a significant performance penalty.

Additionally, many traditional memory allocators are not optimized to handle fragmentation which can degrade system performance over time, especially in long-running applications or applications with dynamic memory allocation patterns, as the allocator has to work harder to find contiguous free blocks or to recombine fragmented blocks of memory.

This project proposed the development of a scalable memory allocator that attempts to solve some of the above problems that are inherent to memory allocators, especially in a multithreaded environment. It introduces the concept of per thread heaps where every thread is allocated a dedicated heap with a certain number of initial pages. This greatly aids in scalability as different threads can be serviced by their private heaps without affecting the other heaps. A memory allocation request originating from a certain thread is only serviced by the heap corresponding to that thread. On the other hand, a memory free operation can

attempt to free the memory block allocated by a different thread. The proposed allocator introduces the concept of ownership where the freed memory block is always returned to the original thread's heap. This reduces false sharing as it ensures that memory blocks of different threads are not present in the same heap, hence, not on the same cache line. Moreover, since the allocation of memory is in terms of page sizes, which happen to be cache aligned, it can be guaranteed that the memory blocks of two different threads will never fall in the same page and hence will never fall on the same cache line. This practically eliminates false sharing across threads. Further, the memory allocator actively merges free blocks and unmaps freed pages back to the Operating System, which helps in reducing fragmentation.

Literature Survey

There has been considerable work in the direction of developing different memory allocators. The default and well-known memory allocator that is popularly used is malloc and free. The C Standard Library introduced malloc and free as part of dynamic memory management. These functions were derived from early systems and are now integral to memory allocation on systems that do not rely on automatic garbage collection. The malloc function allocates a specified number of bytes of memory and returns a pointer to the first byte of the allocated block. If the allocation is successful, the pointer can be used to access the memory. If allocation fails, malloc returns NULL. The free function deallocates memory previously allocated by malloc, returning it to the system. This prevents memory leaks by ensuring that the memory is no longer in use.

In terms of Scalable Memory Allocators, the Hoard Scalable Memory Allocator is a state-of-the-art allocator for multithreaded applications. The Hoard allocator addresses the various challenges inherent to scalable memory allocation such as Lock contention, Scalability, fragmentation, concurrency, etc. by employing a scalable approach that minimizes lock contention and reduces fragmentation in multithreaded environments. The paper proposes several techniques that differentiate Hoard from existing memory allocators, as listed below:

- Hoard uses thread-local storage to reduce contention. Each thread maintains its own local memory pool, minimizing the need to access global memory. This technique allows threads to allocate and deallocate memory from their own pools, reducing the frequency of synchronization operations across threads.
- The Hoard allocator minimizes lock contention by using fine-grained locking and local pools. Threads allocate memory from their local pool, and when a thread runs out of space in its local pool, it may access a global pool. The global pool is protected by a lock, but contention is minimized because it is rarely accessed by threads compared to other allocators that heavily rely on global locks.
- Hoard employs a hierarchical memory pooling approach, where different pool sizes are dynamically allocated based on the size of the requested memory block. This helps manage fragmentation by ensuring that memory is grouped in sizes that are appropriate for the application's allocation patterns.
- Hoard includes an efficient strategy for managing both small and large memory blocks. Small blocks are allocated from thread-local pools, while larger blocks are handled by the global pool. This technique optimizes the allocator for both fine-

grained memory requests (common in many small objects) and coarse-grained requests (for large data structures).

- The Hoard allocator aims to minimize fragmentation by using efficient memory block management techniques. The hierarchical pooling structure and the use of localized memory pools help reduce both external and internal fragmentation.
- Hoard uses a slab allocator design for certain memory management tasks. This approach improves performance by keeping a pre-allocated set of memory blocks (slabs) of a given size, reducing the need for frequent allocation and deallocation. It also reduces fragmentation by grouping similar-sized allocations.

Taxonomy of Memory Allocators

In general, the different memory allocator algorithms in Literature can be categorized into the following categories:

- **Serial Single Heap:** Here, only a single processor may access the heap at any point in time as the global heap is typically protected through a single lock which serializes memory operations. This leads to considerable lock contention in multithreaded applications which affects scalability and hence are not very suitable for multithreaded applications. As it involves a single heap, they actively induce false sharing which can be a significant bottleneck for performance. Some known Serial single Heap Allocators are the default allocators in Solaris and Windows NT/2000.
- **Concurrent Single Heap:** Here, the heap is implemented as a concurrent data structure such as a concurrent B-tree and hence many processors can simultaneously operate on one shared heap. As it involves a single heap, they actively induce false sharing which can be a significant bottleneck for performance. The concurrent data structure also requires the use of many locks and atomic update operations which can end up being quite expensive.
- **Pure Private Heaps:** Here, every processor has its own per-processor heap that it uses to service the memory operations which can be better suited for scalability. This is purely private because every processor never accesses any other heap for any memory operation. Memory allocated by one thread can be freed by another thread. This freed memory is placed in the second thread's heap thereby passively inducing false sharing. Pure private heaps are used in STL's pthread_alloc and Cilk.
- **Private Heaps with Ownership:** Here, every processor has its own heap but the memory is returned to the owner processor's heap on a free operation thereby reducing false sharing scenarios while being scalable. MT-malloc, Ptmalloc and LKmalloc are some of the memory allocators that use private heaps with ownership. The proposed allocator implemented as part of this project, namely, My_mem_alloc falls under this category of having private heaps with ownership. It attempts to scalably support multithreaded applications by reducing false sharing and fragmentation.
- **Private Heaps with Thresholds:** Here, every processor has its own heap which can hold a limited amount of free memory. When a per-processor heap has more than a certain amount of free memory (the threshold), some portion of the free memory is moved to a shared heap. They can passively induce false sharing as pieces of the same cache line can be recycled. As long as the amount of free memory does not exceed the threshold, pieces of the same cache line spread across processors will be

repeatedly re-used to satisfy memory requests, resulting in false sharing scenarios. DYNIX kernel allocator and Hoard are some of the Memory allocators that employ private heaps with thresholds.

Proposed Idea

As part of the research, experimentation and development of the Scalable Memory Allocator for OpenMP Programs, 3 different incremental versions of the Memory Allocator were developed and experimented with, which are detailed as follows:

1. **Sequential Memory Allocator (Single-threaded only implementation)**

   This was the very first version of the Memory Allocator developed which supported only single-threaded client programs. The main objective of this allocator was to develop and finalize some of the core ideas and features inherent to the Memory Allocator, listed as follows:

   a. A **Book Keeping Node** to maintain the metadata information of the memory block it is associated with, like the size of the memory block, and whether it is free or allocated. It also keeps the pointers to the next and previous book keeping nodes and hence is implemented as a doubly linked list of book keeping nodes.
   b. A **Free List node** to maintain information of the memory blocks that are not currently allocated and hence can be used to service a future memory allocation request. These nodes contain pointers to the book keeping nodes of the free memory blocks. They also contain pointers to the previous and next free list nodes, hence the free list is a doubly linked list of free list nodes.
   c. One important consideration was regarding the **Alignment** of the memory blocks. It was made sure that any memory request is first aligned to the nearest word boundary to facilitate efficient load/store operations for this memory block. Further, it was decided that every request to the operating system for memory (an mmap call) will be in terms of page sizes alone. This was done keeping in mind that an mmap call is a system call which can have substantial latency. Hence, rather than making repeated mmap system calls for smaller sized memory, it would be prudent to get a larger sized memory block which can then be internally managed to service future memory requests. Therefore, on a memory allocation request, the requested size is aligned to the nearest page boundary and then an mmap call is made for the deduced number of pages.
   d. On a **Memory Allocation request**, the requested size is first aligned to the word boundary as explained above. The free list is then searched through to find a free block (if, any) that can service this request. The best fit algorithm is implemented to find such a free block; therefore, it searches for the smallest free memory block that is large enough to service the given memory request.
      i. If such a memory block is found, a **split** operation is performed to split the block into 2 blocks, one of the requested size, and another with

the remaining memory (if the remaining memory is not sufficiently large, that is it is smaller than the size of the book keeping node, the split operation is not performed). The newly split block is added to the front of the free list keeping in mind locality and the allocated free block is removed from the free list. The book keeping nodes are created and updated accordingly.

ii. If no such memory block is found in the free list, then there is a requirement to allocate memory by requesting the Operating System. This is done through an **mmap** call. Here, the requested size is aligned to the nearest page boundary as explained above and the newly deduced size (which will be a multiple of the page size) is requested as part of the mmap system call. Once this memory is mapped, a similar split operation is performed as before (if required) resulting in a block to address the request and a remaining memory block. The free list and the book keeping nodes are updated accordingly.

e. On a **Memory Free Request**, certain condition checks are performed first, such as, if the passed pointer is NULL, or if the memory block has already been freed. If the above conditions are not met, the actual free operation is performed. The previous and next memory blocks of the memory block to be freed is checked to determine if they are free as well which would allow them to be merged with the current block. This **merge** operation handles a variety of cases and performs the required updates to the book keeping nodes and the free list. Once the merge operation is done, it is checked to see if the memory block to be freed exceeds the page size and is such that the starting memory address of this block aligns with a page boundary. If these conditions are met, this memory block can be unmapped using the unmap system call. First, a split operation is performed to split any residual parts of the memory block after dividing the size by the page size (as the unmap operation is done only in terms of pages). Then, the unmap system call is made to unmap the deduced number of pages.

f. Further, functions to display the free list and the list of the book keeping nodes along with their memory mapping were developed to aid the development, experimentation and verification of the various features discussed above.

g. With this version of the memory allocator, it was possible to develop, experiment and verify some of the fundamental ideas inherent to the memory allocator being designed. These ideas form the base for the future versions of the memory allocator.

2. **Concurrent Memory Allocator – Serial Single Heap**

The previous version of the memory allocator was such that it was completely sequential and could only support single threaded client applications. The objective of this version of the allocator was to augment the previous version to support multithreaded client applications. The fundamental ideas and features of this version, a concurrent memory allocator is summarized below:

a. The core features of book keeping node, free list, alignment to word and page boundaries, search through best fit algorithm, split and merge operations, map and unmap system calls remain similar to the previous version of the memory allocator described above.
b. The significant modification for this version of the memory allocator is the introduction of a **mutex** variable to lock and unlock access to the data structures and operations of the single heap under consideration. Thereby, this mutex variable provides mutual exclusion to support multithreaded applications performing parallel memory allocation and free operations.
c. On a memory allocation request, a lock operation to the above mutex is performed. The rest of the operations crucial to a memory allocation operation like finding a free block, splitting free blocks, map system call, etc. are performed only after this lock is acquired. The lock is released after the required operations are performed.
d. Similarly, on a memory free request, a lock operation to the above mutex is performed. The rest of the operations crucial to a memory free operation like merging free blocks, unmap system call, etc. are performed only after this lock is acquired. The lock is released after the required operations are performed.
e. Care was taken to handle the locking for various cases such as acquiring the lock only if required, for example, if the memory allocation request was for a size lesser than 0, it can directly return, or if the free request was for a NULL pointer, it can directly return. In such cases, the lock was not even acquired. In certain other cases, care was also taken to unlock the mutex before returning from the function, such as after finding a block in the free list which is a conditional exit or after deducing that the block to be freed is already freed. The base case of returning at the end of functions, after performing all the required operations was explicitly handled such that the lock to the mutex is released before returning from the function.
f. Therefore, the introduction of the mutex and the utilization of its lock and unlock operations to provide concurrent access to a single heap, allows multithreaded applications to perform memory allocation requests and memory free requests using this allocator. However, as the mutex lock prevents multiple threads from accessing the heap simultaneously, at any given point of time, only a single thread can access the heap to service its memory allocation and free requests. Therefore, this version of the memory allocator serializes the memory allocation and free operations.
g. With this version of the memory allocator, it was possible to support multithreaded applications performing memory allocation and memory free requests. Specifically, this version allows for concurrent memory allocation and free by different threads but not parallel memory allocation and free as the mutex serializes the access and operations to the single shared heap.
h. It can be specifically noted that this version of the memory allocator uses a single shared heap to service the memory requests from different threads. This clearly is a bottleneck to performance and this was precisely the concern that was attempted to be handled by the next incremental version of the memory allocator.

3. **Concurrent and Scalable Memory Allocator – Per thread Heap**

   As discussed above, although the previous version of the memory allocator supports multithreaded applications, it serializes the memory operations due to the controlled access to the single shared heap. As part of this incremental version of the memory allocator, various ideas and features were implemented to work around this and better support concurrent, scalable and parallel memory allocation. These are summarized as below:

   a. The core features of book keeping node, free list, alignment to word and page boundaries, search through best fit algorithm, split and merge operations, map and unmap system calls remain similar to the previous versions of the memory allocator described above.
   b. The significant difference is the introduction of a **heap per thread** rather than the single global heap in the previous version of the memory allocator. This heap is dedicated to a single thread and maintains a private linked list of book keeping nodes, private free list and a mutex to control access to this per thread heap. Additionally, other required information like the number of pages allocated to this per thread heap, the last memory mapped address are maintained to support the various operations on the per thread heap. Therefore, there is an array of per thread heaps where every heap corresponds to a single thread and services the memory requests originating from that thread.
   c. It was also decided that each of these heaps will have some **initial free memory mapped** to each of them. The implemented version dedicates 2 pages of initial free memory to every per thread heap (this can be configured and experimented with based on the requirements of different applications). This initial free memory is mapped as part of an initialization call. This initialization is performed only once, as part of the first memory allocation request received by the memory allocator. In order to safely and correctly support this, a mutex is used which provides mutual exclusion in relation to the read, and conditional modify operations performed during init. The init call initializes and grounds the various fields of the per thread heap, maps 2 pages of initial free memory to every per thread heap, updates the free list with this free memory block and updates the number of pages allocated to this per thread heap.
   d. On a **memory allocation request**, the thread number of the thread from which the request originated is deduced. Subsequently, the mutex corresponding to this thread's heap is locked. Once the lock is acquired, the operations inherent to memory allocation are performed as before, such as, finding a free block through the best fit algorithm, mapping memory through mmap system call, split operation if required, etc. Finally, once the required memory is allocated, the mutex of this thread's heap is unlocked.
   e. Similarly, on a **memory free request**, the thread number of the thread from which the request originated is deduced. It is to be noted here that there is one specific case that needs to be handled in relation to memory free. As the programmer is dynamically allocating memory on the heap, the logical view

of the programmer is that every thread can access this dynamically allocated memory, irrespective of the thread that allocated it. Therefore, specifically, in regards to the memory allocator, any thread can free a dynamically allocated memory. This had to be explicitly taken care of by the memory free operation in this version of the memory allocator. First, the current thread's heap is searched through to verify if it contains the memory to be freed. As before, this is done after acquiring the mutex for this thread's heap. If not found, this mutex is first released and the rest of the per thread heaps are searched through to determine where the memory block to be freed is present. As before, this is done after acquiring the lock to the mutex of the per thread heap under consideration. Once the memory block is found on a particular per thread heap, the operations inherent to the memory free are performed as before, such as, merging free blocks, splitting and unmapping if required, etc. One explicit condition taken care of is checking the number of mapped pages against the initial mapped pages. The unmap operation is done only if after unmapping, the remaining pages is still greater than or equal to the initial number of pages (here, 2). This way, it is guaranteed that every per thread heap has at least 2 pages mapped to it at any point in time.

f. The design of a private heap per thread was due to several reasons. One of them was to aid speed and scalability as every thread has a dedicated heap which can be used to service its memory requests without interfering with the other threads or their heaps. The lock contention is also significantly reduced as a thread only attempts to acquire the mutex for its per thread heap apart from the free operation check detailed above.
g. The design also considered various aspects related to False Sharing, an issue inherent to and very prominent in memory allocation for multithreaded applications. As the memory mapping for the per thread heaps is always done in terms of pages, it can be guaranteed that no two per thread heaps will have memory allocated on the same page. More specifically, as the pages are cache aligned, it can also be guaranteed that no two per thread heaps will have memory allocated on the same cache line. This significantly reduces and practically eliminates false sharing across threads. Moreover, when a memory block is freed, the free block is maintained in the original thread itself, rather than moving it to the thread freeing the block. This reduces false sharing as the movement of free blocks could make it such that a free block moved to a new thread is adjacent to a memory block of the original thread, giving rise to false sharing scenarios.
h. Additionally, the design considered various aspects related to fragmentation and attempted to keep the fragmentation as low as possible. The maintenance of a free list combined with the best fit algorithm search through the free list helps to allocate memory from previously free blocks rather than making a new mmap call. The merge operation also significantly aids low fragmentation by merging adjacent free blocks. Further, the unmap operation is performed based on the remaining number of pages and actively returns memory to the Operating System.

i. This 3rd version of the memory allocator with per thread heaps is the one used for all experiments performed and for generating all the required results.
j. This 3rd version of the memory allocator is referred to as **"My_mem_alloc".**

Note: Each of the above 3 versions implement and expose the following functions:

1. *void* my_mem_alloc(size_t size)*: This is the function to allocate memory of the mentioned size
2. *void my_mem_free(void* ptr)*: This is the function to free memory pointed by the mentioned pointer
3. *void display_free_list()*: This is a helper function to display the current contents of the free list
4. *void display_mem_map()*: This is a helper function to display the current memory mapping, specifically the book keeping nodes and their contents

Experimental Setup

All the experiments were performed and the results were gathered on the ***crunchy2 CIMS*** machine which has the following configuration:

Architecture	x86_64
Number of CPUs	64
Number of Sockets	4
Number of Cores per Socket	8
Number of Threads per Core	2
L1d Cache	1 MiB (64 instances)
L1i Cache	2 MiB (32 instances)
L2 Cache	64 MiB (32 instances)
L3 Cache	48 MiB (8 instances)
Page Size	4K bytes
Cache Alignment	64 bytes
L1d Cache Line Size	64 bytes

In order to verify the implemented functionalities and features of the developed memory allocator (the 3 versions explained above), sanity clients for the 3 versions were designed. This performs different Memory Allocation and Memory Free requests and checks for various cases. The sanity client is named as ***client_mem_alloc.cpp*** and is placed within the folder of each of the implemented versions of memory allocator. The sanity client for the third finalized version is an OpenMP program performing memory allocation and free requests from different threads. A makefile is also provided within the folder for each of these versions.

Further details regarding the Source Code folder hierarchy, Benchmark programs folder hierarchy, steps and commands to execute the different versions and the benchmarks are detailed in the attached ***readme.txt*** file.

Experiments and Analysis:

In order to compare the different memory allocators, namely Malloc, Hoard and My_mem_alloc, various benchmark programs were developed as explained below:

1. **Speed:** In order to compare the different memory allocators for speed, single threaded and multithreaded benchmark programs were developed under the following categories:
    a. **No Malloc or free:** This benchmark program does not perform any malloc or free operation and just performs a few basic arithmetic operations within a loop. The single threaded version uses a single thread whereas the multithreaded version uses 8 threads simultaneously performing these operations.
    b. **CPU bound:** This benchmark program performs significantly greater number of arithmetic operations as compared to memory allocation and memory free operations, hence is CPU bound. Each thread dynamically allocates an integer array of 100 elements, performs various arithmetic operations and then finally frees the integer array. The single threaded version uses a single thread whereas the multithreaded version uses 8 threads simultaneously performing these operations.
    c. **Memory bound:** This benchmark program performs a substantial number of memory allocation and memory free operations, hence is Memory bound. Each thread allocates a double pointer array of large size. Each element of this double pointer array is initialized in a loop with a dynamically allocated double array of large size. A second loop is used to free the double arrays and then finally the double pointer array is freed. The single threaded version uses a single thread whereas the multithreaded version uses 8 threads simultaneously performing these operations.

2. **Scalability:** The scalability benchmark program is developed such that every thread allocates 100000/t (where 't' is the number of threads) 12-byte objects (3 integers) and then frees the allocated objects. This benchmark program is then executed for different number of threads, ranging from 1 to 14.

3. **False Sharing Avoidance:** With regards to False Sharing, two benchmark programs were developed for the following categories:
    a. **Active False Sharing:** In this benchmark program, every thread allocates 8 bytes (2 integer) objects, performs a large number of write operations on these objects and then frees the objects. This benchmark program uses 8 threads. Since these 8 threads simultaneously allocate 8 bytes, all the requested memory (8*8 = 64 bytes) can fit into a single cache line of 64 bytes. Therefore, this benchmark program verifies the active false sharing scenario.
    b. **Passive False Sharing:** In this benchmark program, the sequential part of the program allocates 8 integer objects which are then handed over to each of the 8 threads. Every thread frees one of the integer objects, then allocates a new integer object, performs a large number of write operations on it and

then finally frees the integer object. The nature of this program is such that it can have false sharing at the beginning given that the 8 integer objects can be contiguous and are part of the same cache line. Once these objects are freed, the memory allocator can passively induce false sharing for future mallocs if they are allocated using the memory freed by these objects, thereby falling in the same cache line as the objects of other threads. Therefore, this benchmark program verifies the passive false sharing scenario.

4. **Fragmentation:** The Fragmentation benchmark program is developed such that every thread allocates an integer pointer array of large size, each pointer is then initialized to a dynamically allocated integer array of large size (say, size 's1'). Then every alternate element of the integer pointer array is freed, after which a new integer pointer array is allocated, where every pointer is then initialized to an integer array of the same size as before – 's1'. Therefore, half of these new memory requests can be satisfied by the memory freed from the previous array. If not handled properly, the memory allocator can map and allocate fresh memory rather than reusing these freed blocks, leading to high fragmentation. Therefore, this benchmark program verifies the fragmentation scenario for the different memory allocators.

- All the benchmark programs are provided under appropriately named folders under the *comparison_testing* folder hierarchy.
- The execution time calculation for the different programs was performed using the *omp_get_wtime()* utility.

Details regarding running the benchmark programs for the different memory allocators:

- The benchmark programs are linked with the C standard library (cstdlib) to invoke the default malloc and free calls.
- For the Hoard memory allocator, the source code for Hoard is first downloaded from the Github Repository (https://github.com/emeryberger/Hoard) which is then built to generate the libhoard.so shared object file. This is then linked with the object file of the benchmark program to generate the required executable.
- For the My_mem_alloc version of the memory allocator, the custom_mem_alloc.h header file provided in the 3rd version (Per thread Heaps) is used to provide the required interface. The custom_mem_alloc.o generated for the Per thread Heaps version is linked with the benchmark program to generate the required executable.

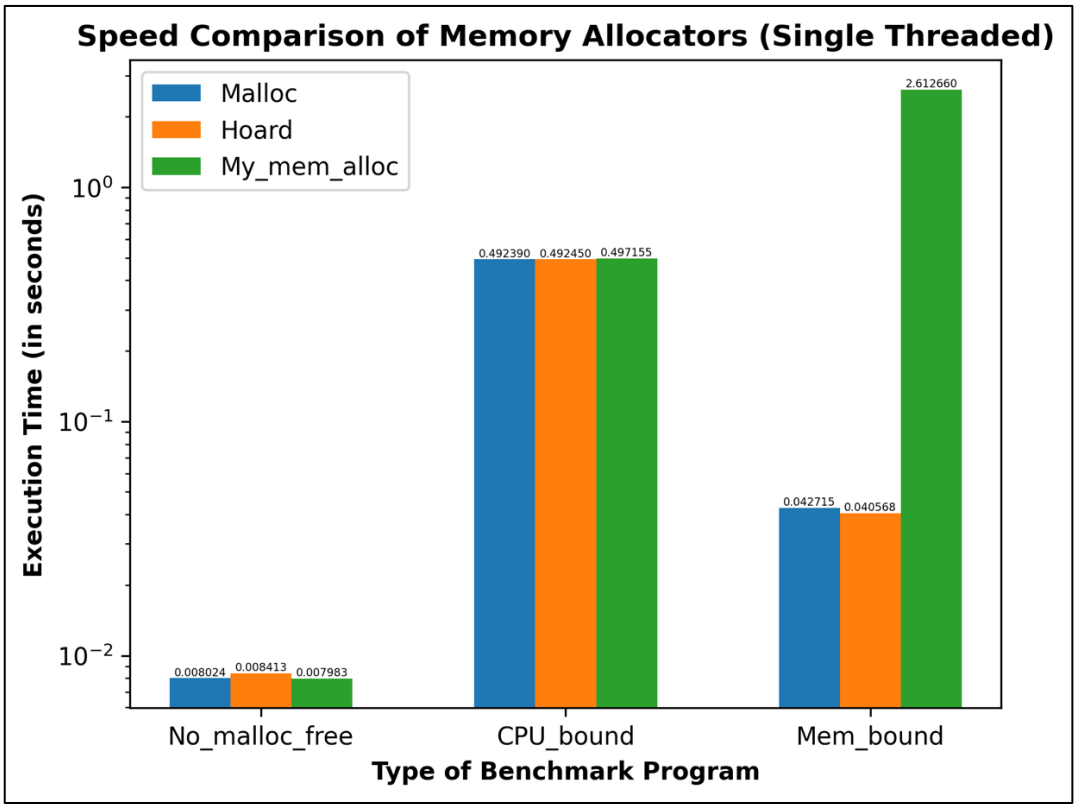
Results

Malloc, Hoard and My_mem_alloc memory allocators were compared based on the following metrics using the benchmarks detailed above:

1. **Speed:**

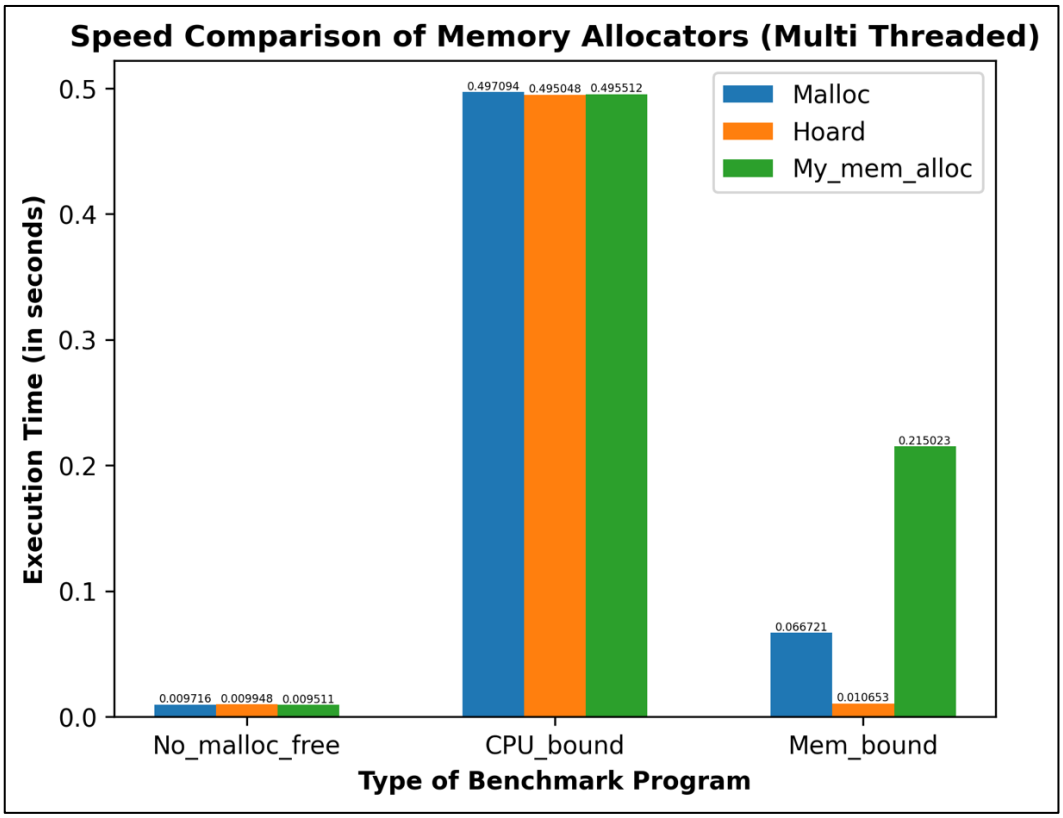
   In order to compare the different memory allocators for speed, the execution time in seconds was used as the metric.

The figure below summarizes the speed comparison of the memory allocators for the single threaded benchmarks:



From the above figure it can be seen that the execution times of the different memory allocators are very similar for the No_malloc_free and CPU_bound benchmarks, which is the expected scenario as these benchmarks do not have a significant number of memory operations. For the Mem_bound benchmark, it can be seen that execution times are very similar for Malloc and Hoard whereas the execution time of My_mem_alloc is relatively higher. This increased execution time for My_mem_alloc comes from the additional operations of maintaining a thread per heap, mapping and maintaining some initial number of pages to each heap, performing a best fit algorithm search to find the free block and merging free blocks.

The figure below summarizes the speed comparison of the memory allocators for the multithreaded benchmarks, that were executed with 8 threads:

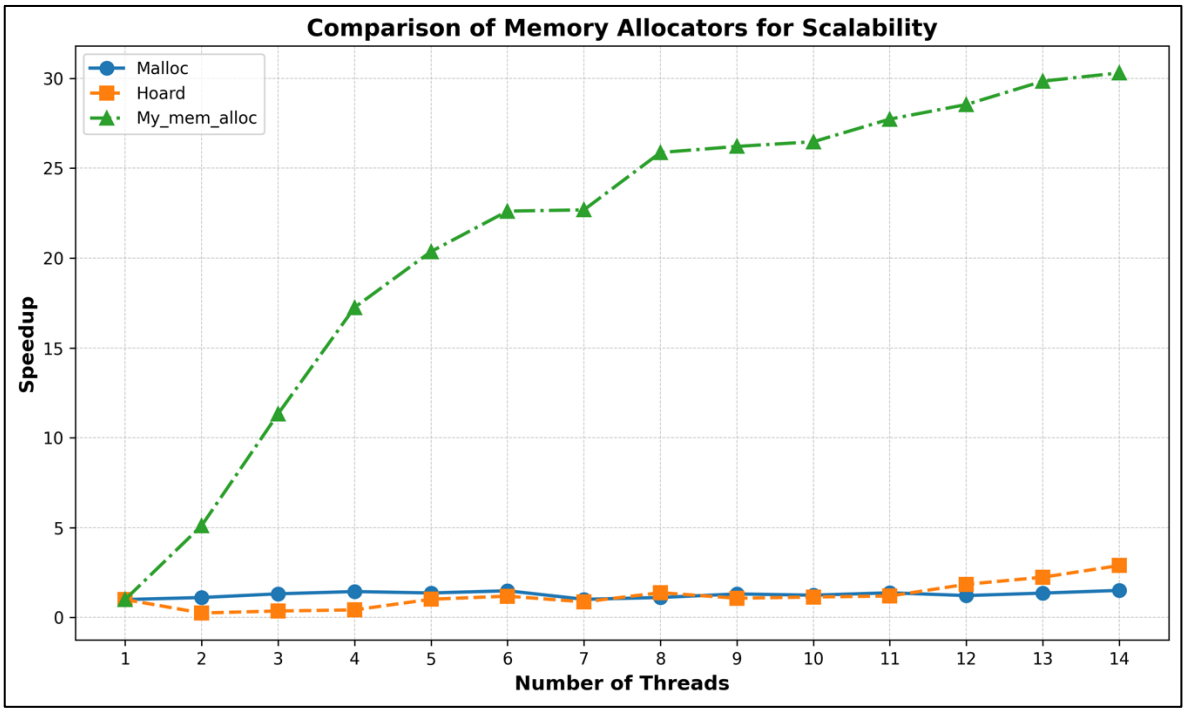


From the above figure it can be seen that the execution times of the different memory allocators are very similar for the No_malloc_free and CPU_bound benchmarks, which is the expected scenario as these benchmarks do not have a significant number of memory operations. For the Mem_bound benchmark, it can be seen that execution time is the least for Hoard, second least is for Malloc and finally the highest is for My_mem_alloc denoting that Hoard exhibits good performance in terms of execution time for Memory Bound benchmarks. The increased execution time for My_mem_alloc comes from the additional operations of maintaining a thread per heap, mapping and maintaining some initial number of pages to each heap, performing a best fit algorithm search to find the free block and merging free blocks.

2. **Scalability:**

In order to compare the different memory allocators for scalability, the calculated speedup in relation to one thread [Speedup for n threads = (Execution time for 1 thread)/(Execution time for n threads)] was used as the metric.

The figure below summarizes the scalability comparison of the memory allocators for the scalability benchmarks:

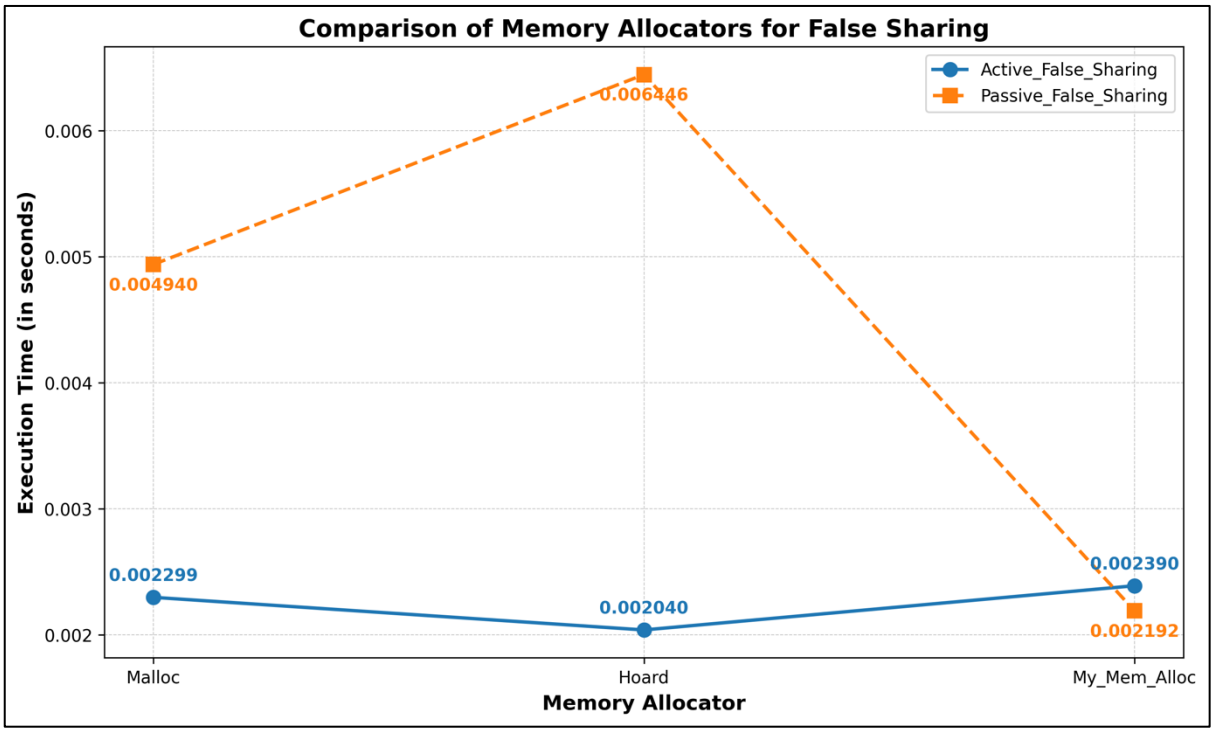


From the figure above, it can be seen that the speedup increases for the My_mem_alloc memory allocator with the increase in the number of threads. On the other hand, the speedup for the malloc and hoard memory allocators increases very negligibly with the increase in the number of threads. A general rule of thumb is that as the number of threads increases, the performance of the memory allocator must scale, ideally linearly, with the increase in the number of threads to ensure scalable application performance. Here, the speedup is the indicator for performance. Although the speedup increase for My_mem_alloc is not completely linear (in fact it is better than linear speedup in most of the cases), the speedup increase seen with the increase in the number of threads in the above graph denotes that it is the most scalable memory allocator, quite considerably as compared to the Malloc and Hoard Memory Allocators. This could be due to the fact that even with the increased number of threads, every thread still has access to a private per thread heap to service the memory requests. As these per thread heaps are logically independent, they can better service the memory requests simultaneously coming from the different parallel threads, even with an increase in the number of threads.

3. **False Sharing Avoidance:**

In order to compare the different memory allocators for false sharing avoidance, the execution time in seconds was used as the metric.

The figure below summarizes the false sharing comparison of the memory allocators for the false sharing benchmarks which were executed with 8 threads:



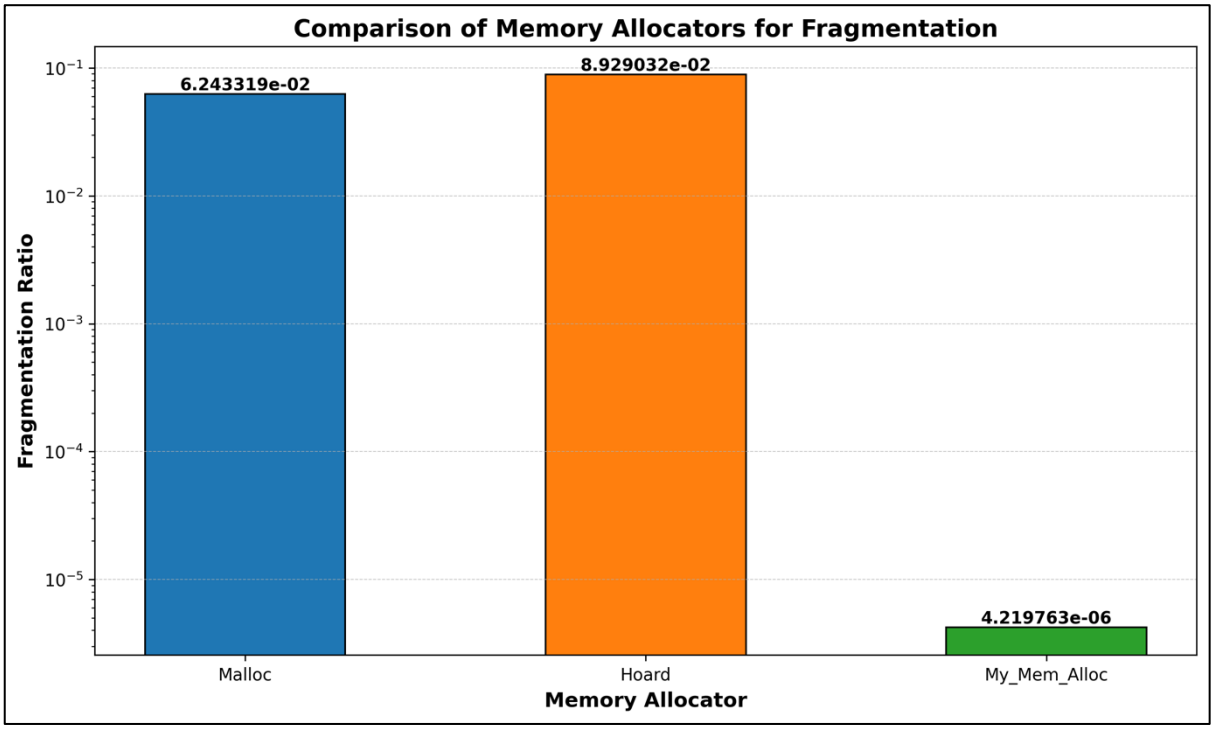
From the above figure, it can be seen that the My_mem_alloc memory allocator has the best performance (execution time) for both the Active False Sharing and Passive False Sharing benchmarks. In general, a memory allocator should not introduce false sharing of cache lines in which different threads inadvertently share data on the same cache line. As the design of the My_mem_alloc memory allocator is such that there are private heaps per thread from which the memory requests generated from that thread are serviced and the fact that the memory is mapped only in terms of pages (which are cache aligned), it ensures that memory on two different per thread heaps are never in the same page and hence never in the same cache line. Therefore, the My_mem_alloc memory allocator practically eliminates false sharing which is also corroborated by its performance for the false sharing benchmarks. On the other hand, the other two memory allocators, namely Malloc and Hoard have comparable performance to the My_mem_alloc memory allocator for the Active False Sharing benchmark but a relatively worse performance for the Passive False Sharing benchmark, denoting that Malloc and Hoard do not completely avoid false sharing, especially for the Passively induced False Sharing scenario.

4. **Fragmentation:**

In order to compare the different memory allocators for fragmentation, the fragmentation ratio was used as the metric. The fragmentation ratio can be defined as the maximum amount of memory allocated from the operating system divided by the maximum amount of memory required by the application. The maximum amount of memory required by the application can be deduced by analyzing the

benchmark program. On the other hand, the maximum amount of memory allocated from the operating system cannot be deduced very easily. This is more of an issue considering how it is not possible to tap into the implementations of malloc and hoard and add additional markers to calculate this information. As the comparison needs to be performed uniformly across the memory allocators it was decided to use a way of estimating this number. The "*cat /proc/<pid>/maps | grep heap*" linux command gives the memory range for the heap area of the process under consideration (Here pid is the process id of the executable under consideration which can be deduced from the *"ps aux | grep <executable_name>"* command). By adding an explicit sleep for a large number of seconds at the end of the program, the process can be guaranteed to be in execution and hence the range of the heap can be deduced as explained above. This range of heap (say, start-end) is then used to deduce the number of bytes by performing a simple subtraction between end and start. This is used to estimate the maximum amount of memory allocated from the operating system which is then used to calculate the fragmentation ratio.

The figure below summarizes the fragmentation comparison of the memory allocators for the fragmentation benchmarks which were executed with 8 threads:



From the above graph, it can be seen that the My_mem_alloc memory allocator has a very low fragmentation ratio as compared to the malloc and hoard memory allocators denoting that the My_mem_alloc memory allocator is such that it aids low fragmentation. This can be due to the fact that every thread has a private heap to service the memory requests where the free blocks are searched through in a best fit manner; the merge free block operation which merges adjacent free blocks (this can be crucial in reducing fragmentation) and the unmap operation which actively releases mapped memory to the operating system when memory objects are freed.

Conclusion

- This project introduced a scalable memory allocator that uses the concept of per-thread heaps with ownership where a heap is dedicated to a particular thread and is initialized with a certain number of pages. It implements various features such as splitting and merging of free blocks, best fit algorithm to find free blocks, unmapping of freed memory, returning free memory to the original thread, aligning memory block requests to word boundaries and aligning memory map requests to page sizes to eliminate false sharing.
- It also introduces various benchmark programs under the categories of Speed, Scalability, False Sharing Avoidance and Fragmentation which are used to compare and contrast the proposed memory allocator (My_mem_alloc) with the Malloc and Hoard memory allocators. The generated results are promising and show that the proposed memory allocator (My_mem_alloc) exhibits better results especially for Scalability, False Sharing avoidance and Low Fragmentation as compared to the Malloc and Hoard memory allocators.
- Admittedly, there is substantial scope for future work. One of the main drawbacks of the proposed memory allocator seems to be its speed in relation to the Malloc and Hoard memory allocators. This could be due to the nature of some of its inherent features like initializing per thread heaps with some pages, merging of free blocks, best fit algorithm to find free blocks, unmapping freed blocks, etc. In this direction, attempts could be made to improve the speed while keeping the other benefits intact, for example, experimenting with other find free block algorithms like first fit, last fit; unmapping based on some memory usage statistics, compaction etc.

References

- J. S. Jones, M. S. Bhatia, and D. M. Tullsen, "Hoard: A scalable memory allocator for multithreaded applications," ACM Transactions on Computer Systems (TOCS), vol. 20, no. 1, pp. 1-15, 2002.
- M. J. Freedman and S. G. Zeldovich, "The effect of memory allocators on parallelism in multi-core systems," Proceedings of the 5th ACM SIGPLAN Symposium on Memory Management (SAMOS), 2014.
- G. J. W. Gable, S. B. Miller, and P. R. D. King, "Optimizing memory management in multithreaded applications: The importance of locality," Concurrency and Computation: Practice and Experience, vol. 21, no. 7, pp. 893-914, 2009.
- P. J. Lee, M. S. Bhatia, and W. D. Young, "Reducing fragmentation and contention in multithreaded systems through hierarchical memory allocation," ACM Transactions on Computer Systems (TOCS), vol. 32, no. 2, pp. 1-27, 2014.
- B. Bigler, S. Allan, and R. Oldehoeft. Parallel dynamic storage allocation. *International Conference on Parallel Processing*, pages 272–275, 1985.
- T. Johnson. A concurrent fast-fits memory manager. Technical Report TR91-009, University of Florida, Department of CIS, 1991.
- T. Johnson and T. Davis. Space efficient parallel buddy memory management. Technical Report TR92-008, University of Florida, Department of CIS, 1992.

- A. K. Iyengar. *Dynamic Storage Allocation on a Multiprocessor*. PhD thesis, MIT, 1992. MIT Laboratory for Computer Science Technical Report MIT/LCS/TR–560.
- A. K. Iyengar. Parallel dynamic storage allocation algorithms. In *Fifth IEEE Symposium on Parallel and Distributed Processing*. IEEE Press, 1993.
- R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, Santa Fe, New Mexico, Nov. 1994.
- P. Larson and M. Krishnan. Memory allocation for long-running server applications. In *ISMM*, Vancouver, B.C., Canada, 1998.
- P. E. McKenney and J. Slingwine. Efficient kernel memory allocation on shared-memory multiprocessor. In USENIX Association, editor, *Proceedings of the Winter 1993 USENIX Conference: January 25–29, 1993, San Diego, California, USA*, pages 295–305, Berkeley, CA, USA, Winter 1993. USENIX.