15618 – Parallel Computer Architecture and Programming Project

Vaishnavi Ramsali and Vignesh Harish Kashyap

VVMalloc

Parallel and mostly Lock-Free Version of Malloc

SUMMARY

The project involves implementation of a custom thread safe malloc that uses lock-free queues, spinlocks and thread local storage and is scalable when multiple threads are run. The custom implementation has elements drawn from multiple publications, TCMalloc, hoard allocator and has been tested extensively against different trace files from the 15-213 malloc lab. We run this allocator on the shark machines and the GHC machines to compare how different configurations of the machine impact our design.

BACKGROUND

The intention behind delving into a topic like dynamic memory allocation is the interest towards computer systems and the fact that it is at an intersection of parallel computing as well.

In a multi-core environment, most processes are multi-threaded to reap the benefits of parallel computing. However, when it comes to dynamically allocating memory, we must deal with a global heap in most programs which is the major source of contention. In our attempt at implementing a parallel malloc, we try to reduce the repetitive access to the global heap thus reducing contention.

The 213 malloc lab was something we found to be a very good starter code to develop our version of thread safe parallel malloc since the infrastructure checks for correctness and has sufficient trace files for us to draw conclusions.

The 213-lab implementation involved having a segregated list for different size requests to reduce fragmentation and improve throughput. However, our 213 malloc implementation caters for a uniprocessor system with the free lists and the mini blocks being global in nature. Making this implementation thread safe would require us to use locks which would have a negative impact in terms of performance as this is blocking in nature. Considering this, we had to rewrite the algorithm to reduce the repetitive access to global data structures and potentially in a lock free manner to reduce this overhead. We studied different allocators which do a fairly good job in order to seek an insight into the different ways of designing our allocator – VVMalloc.

Hoard Allocator

Hoard maintains per-processor heaps and one global heap. When a per-processor heap's usage drops below a certain fraction, Hoard transfers a large, fixed size chunk of its memory from the per-processor heap to the global heap, where it is then available for reuse by another processor. [1]

The Hoard Allocator mainly addresses 3 negative aspects of conventional memory allocations being

- Contention - When multiple threads allocate and deallocate memory, most allocators serialize this which is the primary bottleneck. This causes contention and the hoard allocator eliminates this bottleneck.
- False Sharing - Threads on different CPUs can end up with memory in the same cache line. Say two processors access different parts of the same cache line. Although they do not modify each other's data, the fact that it is being shared can cause the processor to reload

the entire line each time which takes 100s of cycles more. This is false sharing and is designed to be prevented using a Hoard Allocator.

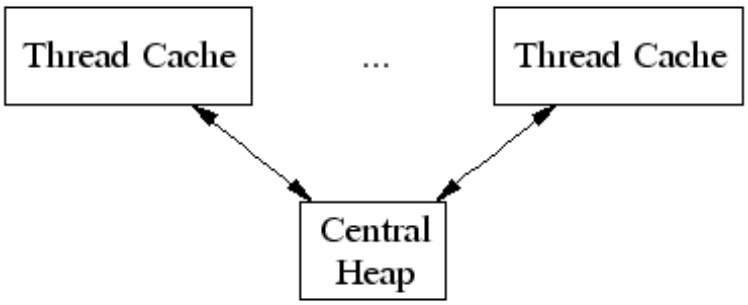
- Blowup - This is a scenario when the memory consumption is much more than the actual required memory. This can also be prevented using a Hoard allocator.

JEMalloc

This is another general-purpose memory allocator which emphasizes on avoiding fragmentation and allows for concurrency which can be scaled up. JEMalloc uses different arenas and thread local caches to avoid contention and uses red black trees and an optimized slab allocator to avoid fragmentation. [2]

How does TCMalloc work?

TCMalloc assigns each thread a thread-local cache. Small allocations are satisfied from the thread-local cache. Objects are moved from central data structures into a thread-local cache as needed, and periodic garbage collections are used to migrate memory back from a thread-local cache into the central data structures. [3]



TCMalloc performs the action of allocation based on the size of the request. If the requested data is below a certain threshold, then the allocator services it from the local thread cache and if this

request is above a certain threshold, then it services it from the central heap as show in the above diagram.

This allocation of small sized objects is somewhat closely related to the 213 implementation of segregated lists. TCMalloc allocates a class to an object based on the range of size it falls under. For instance, if the size of the object is 963 bytes, it is rounded up to the next available class of 1024 bytes. This is the idea behind a segregated list as well.

For larger object sizes, the same idea is scaled up to pages. The heap managed by TCMalloc consists of a set of pages. A run of contiguous pages is represented by a Span object. A span can either be allocated, or free. If free, the span is one of the entries in a page heap linked list. If allocated, it is either a large object that has been handed off to the application, or a run of pages that have been split up into a sequence of small objects. If split into small objects, the size-class of the objects is recorded in the span. [4] When an object is deallocated, the corresponding span object is looked up. This provides the information regarding the object size regarding which class it belongs to or which page it belongs to in case they are large objects. In case these are small objects, the chunk of memory is added to the free objects list in the linked list of the per thread cache. In case it is above a certain size, then a garbage collector is run from time to time which moves the unused objects to the central free list.

Given these features of TCMalloc, and how closely it associates with the existing implementation of the 213 malloc lab, we decided to develop VVMalloc with TCMalloc as our reference point.

APPROACH

The following section has been developed incrementally with us trying to implement the infrastructure, lock free data structures and the allocator itself incrementally. We ran our allocator on the GHC and shark machines. As the performance on both machines show a similar pattern, we have published our performance results as obtained on the GHC machines.

Approach 1: Global Lock based implementation – Baseline version

In this approach, we use the 213 malloc code as our starter code. The main idea here was to first implement a thread safe allocator by naively locking the global heap and the global data structures using a mutex. Although the 213 code has mini blocks as well as segmented lists, we use a common global lock to ensure that if each thread picks up a trace and validates it, we are still able to obtain the correct results. After having these global locks for the data structures, the next part of it was to emulate the same for calculating utilization and throughput. Utilization was straightforward as compared to the timing calculation whose implementation in the 213's mdriver was not thread safe. So, we went ahead and made the mdriver's timing calculation thread safe and compared the serialized code with the multithreaded locked version. The challenging part of it was to understand the mdriver file itself which took us a few days to know what is going on before we could modify it.

```
bash-4.2$ ./mdriver
Found benchmark throughput 8009 for cpu type Intel(R)Xeon(R)CPUE5520@2.27GHz, benchmark regular
Throughput targets: min=4004, max=7208, benchmark=8009
.....................
Results for mm malloc:
  valid    util     ops   msecs  Kops/s  trace
   yes    78.4%      20     0.012   1702 ./traces/syn-array-short.rep
   yes    13.4%      20     0.004   4986 ./traces/syn-struct-short.rep
   yes    15.2%      20     0.004   4874 ./traces/syn-string-short.rep
   yes    73.1%      20     0.006   3238 ./traces/syn-mix-short.rep
   yes    16.0%      36     0.005   7165 ./traces/ngram-fox1.rep
   yes    73.6%     757     0.639   1186 ./traces/syn-mix-realloc.rep
 * yes    75.4%    5748     0.331  17387 ./traces/bdd-aa4.rep
 * yes    71.3%   87830     5.451  16114 ./traces/bdd-aa32.rep
 * yes    71.6%   41080     3.191  12872 ./traces/bdd-ma4.rep
 * yes    71.8%  115380     7.843  14711 ./traces/bdd-nq7.rep
 * yes    75.4%   20547     1.095  18762 ./traces/cbit-abs.rep
 * yes    78.6%   95276     5.032  18935 ./traces/cbit-parity.rep
 * yes    77.1%   89623     5.195  17250 ./traces/cbit-satadd.rep
 * yes    70.9%   50583     2.433  20787 ./traces/cbit-xyz.rep
 * yes    61.7%   32540     1.704  19097 ./traces/ngram-gulliver1.rep
 * yes    58.3%  127912     5.926  21586 ./traces/ngram-gulliver2.rep
 * yes    59.4%   67012     3.073  21809 ./traces/ngram-moby1.rep
 * yes    59.6%   94828     4.544  20869 ./traces/ngram-shake1.rep
 * yes    95.1%   80000    12.682   6308 ./traces/syn-array.rep
 * yes    92.2%   80000     8.712   9183 ./traces/syn-mix.rep
 * yes    84.7%   80000     6.278  12743 ./traces/syn-string.rep
 * yes    86.5%   80000     6.172  12962 ./traces/syn-struct.rep
16 16     74.3% 1148359    79.662

Average utilization = 74.3%.
Average throughput (Kops/sec) = 14639.
Perf index = 60.0 (util) + 40.0 (thru) = 100.0/100
bash-4.2$
```

Fig 1: Sequential version

```
Found benchmark throughput 8009 for cpu type Intel(R)Xeon(R)CPUE5520@2.27GHz, benchmark regular
Throughput targets: min=4004, max=7208, benchmark=8009
.....................
Results for mm malloc:
  valid    util     ops   msecs  Kops/s  trace
   yes    57.2%      20     0.121    166 ./traces/syn-array-short.rep
   yes     0.4%      20     0.235     85 ./traces/syn-struct-short.rep
   yes     0.0%      20     0.153    130 ./traces/syn-string-short.rep
   yes     6.5%      20     0.082    243 ./traces/syn-mix-short.rep
   yes     0.3%      36     0.099    364 ./traces/ngram-fox1.rep
   yes     5.3%     757     4.486    169 ./traces/syn-mix-realloc.rep
 * yes     0.2%    5748     2.200   2613 ./traces/bdd-aa4.rep
 * yes     1.8%   87830    15.173   5789 ./traces/bdd-aa32.rep
 * yes     0.9%   41080    48.684    844 ./traces/bdd-ma4.rep
 * yes     3.4%  115380    11.836   9748 ./traces/bdd-nq7.rep
 * yes     0.3%   20547     4.986   4121 ./traces/cbit-abs.rep
 * yes     1.2%   95276    35.987   2648 ./traces/cbit-parity.rep
 * yes     1.1%   89623    72.945   1229 ./traces/cbit-satadd.rep
 * yes     0.5%   50583    42.770   1183 ./traces/cbit-xyz.rep
 * yes     0.2%   32540    38.229    851 ./traces/ngram-gulliver1.rep
 * yes     0.8%  127912    58.076   2202 ./traces/ngram-gulliver2.rep
 * yes     0.4%   67012    35.735   1875 ./traces/ngram-moby1.rep
 * yes     0.5%   94828    66.633   1423 ./traces/ngram-shake1.rep
 * yes    81.7%   80000    79.838   1002 ./traces/syn-array.rep
 * yes    47.7%   80000    34.188   2340 ./traces/syn-mix.rep
 * yes     2.5%   80000    58.943   1357 ./traces/syn-string.rep
 * yes     2.3%   80000    35.464   2256 ./traces/syn-struct.rep
16 16      9.1% 1148359   641.686

Average utilization = 9.1%.
Average throughput (Kops/sec) = 1668.
Perf index = 0.0 (util) + 0.0 (thru) = 0.0/100
```

Fig 2: Thread-safe baseline version

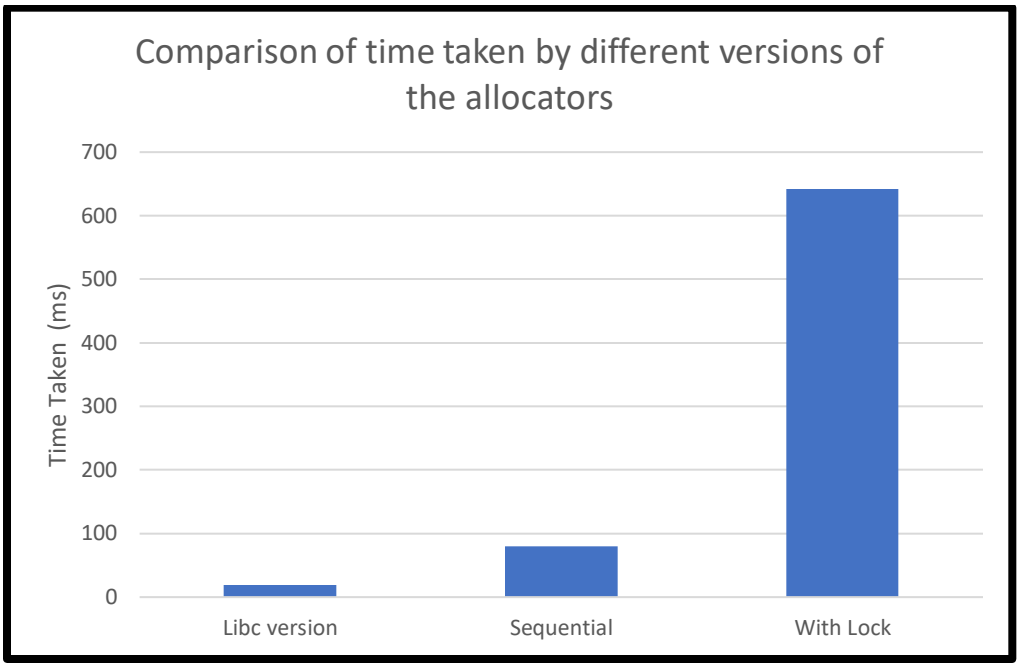


Fig 3: Time taken by different allocators for threads with unequal workload

As we can see from the above graph, the time taken by the memory allocator with a global lock is considerably longer time in the order of 7-8 times it takes to run the sequential version of the code for the same trace files. Therefore, this is not the way to achieve a thread safe dynamic memory allocator as the overhead incurred because of scheduling preemption and contention on the global data structures is immense.

Approach 2: Custom Parallel Malloc

From our first approach, we realized that accessing the list of free blocks (global data structure) by multiple threads for every allocation and free operation was clearly not the right approach as results show. We then re wrote our code by restructuring our algorithm to do the following:

- Our heap is managed in such a way that each thread is initially allotted an independent arena of a minimum of 64kB (unless a request of larger size is made) which is added to

the thread's list of free blocks. The thread's list of free blocks is local in the sense that the pointer to the list is stored in the thread local storage (TLS). Each arena is isolated using an epilogue and prologue such that no coalescing is possible between adjacent arenas or between free blocks belonging to different threads.

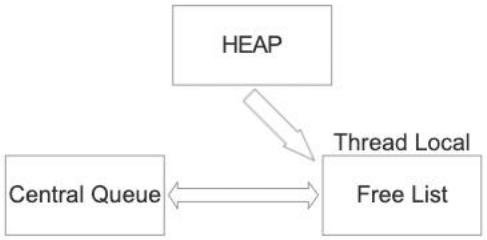


Fig 4: Approach 3 design

- We maintain a segregated central queue to store arenas greater than or equal to 64kB that had been allocated to the process but not been assigned to any thread yet.

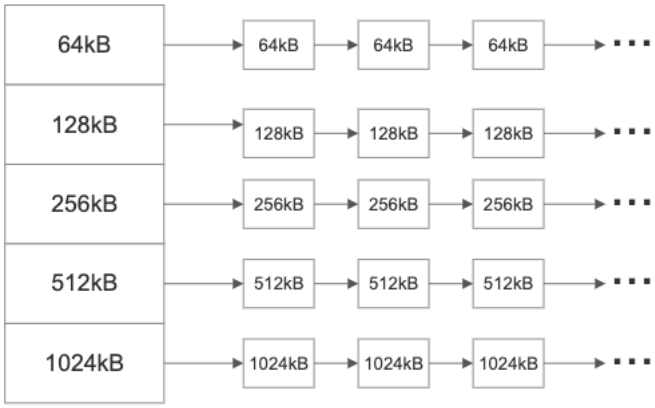


Fig 5: Segregated Central Queue with 64kB being the minimum arena size

- Each thread maintains a list of free blocks with the pointer to the list in the thread local storage (TLS) to avoid false sharing and contention.

- For every allocation request, the thread first checks its free list in the TLS to ensure that the request can be fulfilled.
- In case the request cannot be fulfilled from the list of free blocks in the TLS, the thread requests the allocation of an arena from the central queue.
- When a slice of memory is allocated from the TLS, the allocated section splits itself so that the remaining chunk of memory gets added back to the thread local free list.
- In the scenario that the memory in the central queue is exhausted, the thread requests a heap extension. Based on the size of the request, an adequate block of memory which is either greater than or equal to 64kB is allotted to thread requesting this chunk of memory. We avoid the overhead of storing this newly allocated memory from the heap to the stack list by directly providing it to the thread and resuming normal functionality by letting garbage collection take care of the freed memory into the central stack.
- Every time a free operation deallocates memory, it is returned or appended to the thread local free list. We only perform garbage collection by looking for free blocks in TLS greater than 64kB and adding it to the central stack.
- The other element in our design is having to have a mutex for the central stack which has a global scope. This is needed because multiple threads can access the queue to request a slice of memory.
- We tested this approach on a bunch of traces with unequal workloads with 16 threads working on different traces and obtained the following results in terms of timing.

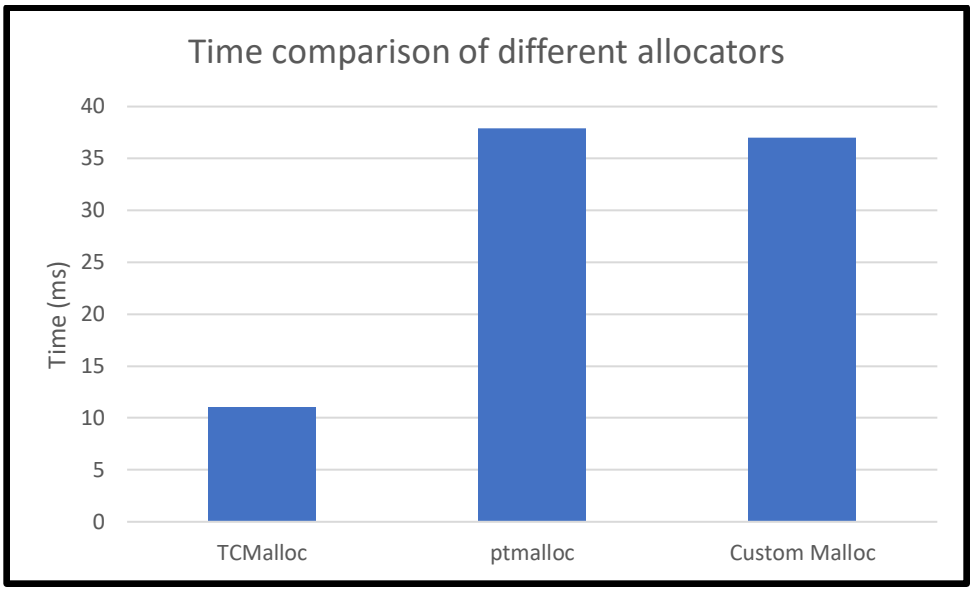


Fig 6: Execution time of different allocators on threads with unequal workloads

1. We can see that with this approach, TCMalloc stands out in terms of performance while our custom malloc is comparable to glib ptmalloc and performs slightly better. The approach is further tested on the following workloads:
2. Workload 1: ~128k operations per thread and the allocation sizes being mostly small and usually less than 32 bytes.

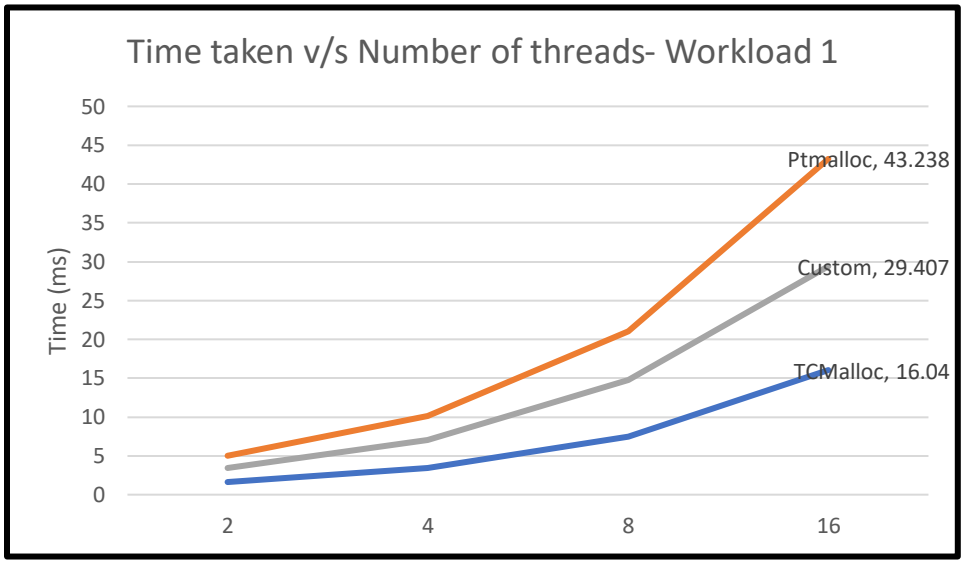


Fig 7: Workload 1 – time taken vs number of threads

3. Workload 2: 80k operations per thread and majority of the allocation requests ranging from sizes between 3kb to 12kb.

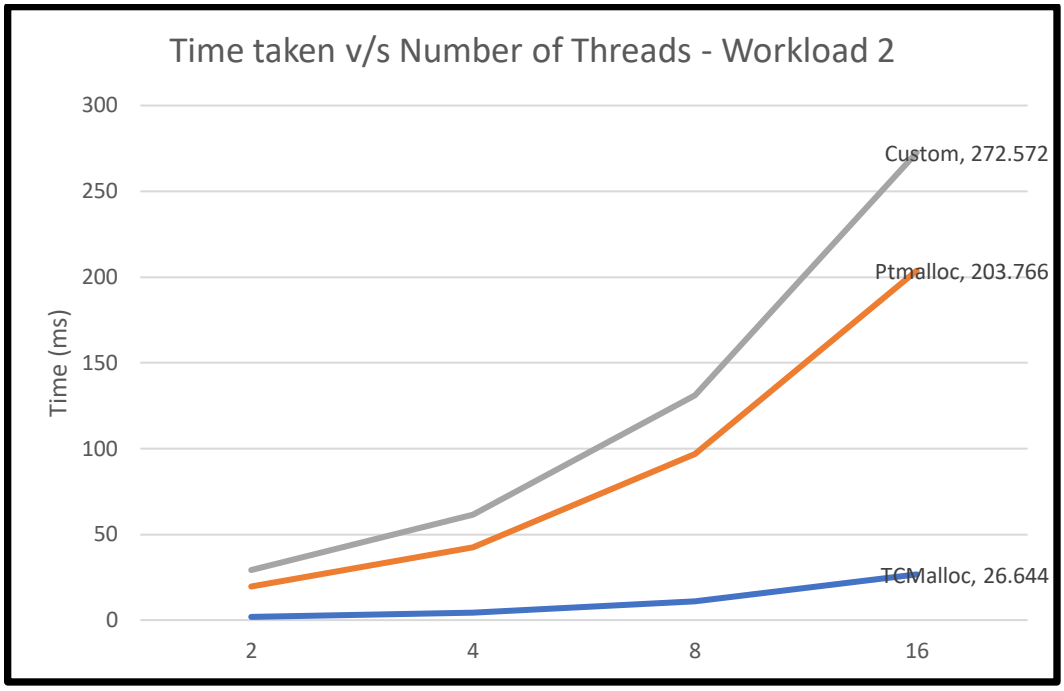


Fig 8: Workload 2 – Time taken vs number of threads

4. Workload 3: 80k operations per thread consisting of allocation requests having a mix of sizes all the way from 4 bytes to 21kB.

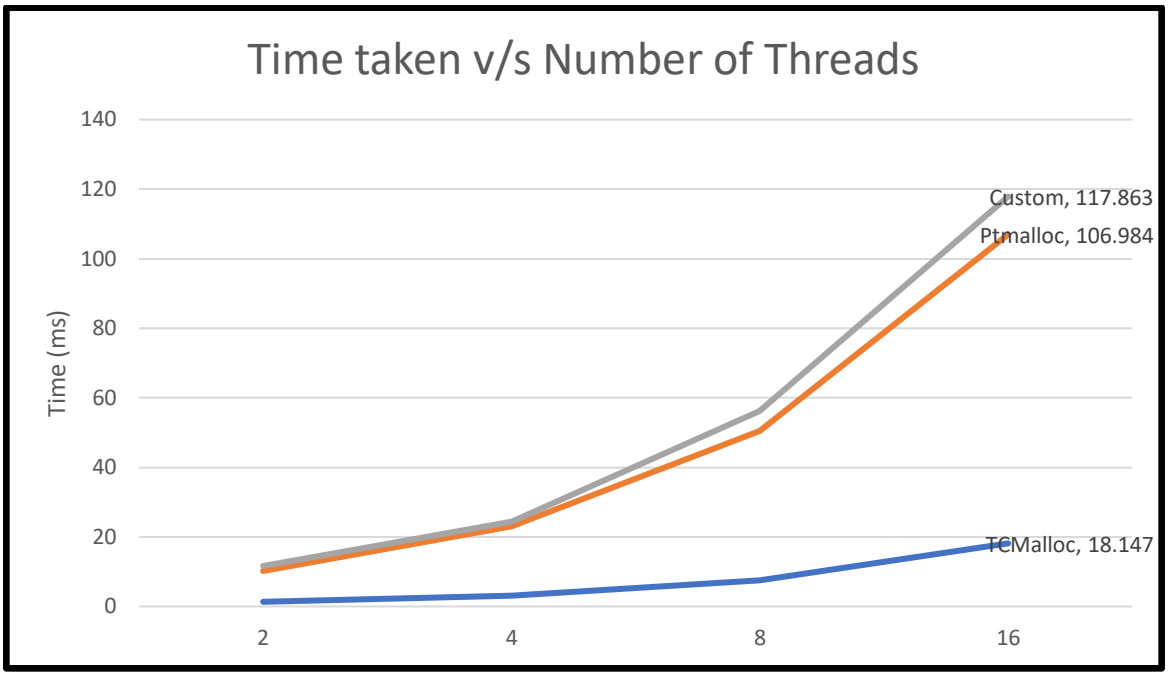


Fig 9: Workload 3 – Time taken vs number of threads

Based on our observations above, we see that our custom malloc performs better than ptmalloc for small allocation requests but takes more time than TCMalloc. For large allocation requests, our custom malloc performs worse than ptmalloc and TCMalloc. For a mix of small and large allocation requests, our custom malloc is comparable with ptmalloc's performance but performs worse than TCMalloc.

To analyze the inconsistency above, we ran perf to profile our code's performance. Based on the perf report results, the spinlock around sbrk to extend the heap causes the maximum overhead. The overhead incurred due to spinlock for the three workloads are as follows:

Workload 1: 14.96%

Workload 2: 93%

Workload 3: 93.38%

Since workload 1 consists of small allocation requests, the threads fully utilize the arena assigned to them, thus rarely requesting pages from the central queue or request a heap extension. This way the threads rarely hit the mutex protecting the central queue or spinlock protecting the heap. However, for workload 2 and 3, the assigned arenas are fully utilized due to larger requests causing the threads to request for more memory from the central queue or the heap thus increasing contention on these global data structures. This can be validated from the perf report where in workloads 2 and 3 have 90% and 93.38% of overhead due to waiting at the spinlock. Thus, the spinlock around the heap causes a major bottleneck. Apart from the spinlock protecting the heap, the mutex protecting the central queue also causes an overhead due context switching.

Approach 3: VVMalloc

In our previous approach, we realized that the two main bottlenecks are caused due to (i) the mutex protecting the central queue of arenas, and (ii) the spinlocks protecting the heap. The mutex can delay other threads from accessing the data structure due to scheduling preemption and page faults whereas the spinlock leads to an additional overhead due to spinning. In this approach we modify our previous approach to implement a lock-free FIFO queue using compare_and_swap (CAS) instead of having a mutex protect the central queue. For our implementation, we followed the *Michael and Scott non-blocking concurrent queues algorithm*. [5]

Here, the head and tail point to a dummy node which is the first node in the list. The tail either points the last node or penultimate node in the list. The dequeuing operation with the help of dummy node ensures that the head and tail never point to a dequeued node. Here we avoid the

ABA problem by using modification pointers which holds the pointer to the node along with count. For consistent values, we recheck the earlier values (pointer and count value) at every step in the sequence of reads to ensure that the earlier values haven't changed.

Enqueue: We use CAS to swing the tail to the next node in the list if it was not already pointing to the last node. This CAS operation continues till the tail->next is NULL. We then perform CAS to try and link the tail to the new node. This process continues till enqueue is successfully done. Once enqueue is successful, we try to swing the tail to the newly inserted node using another CAS operation.

Dequeue: In our approach, we use the dequeue method to the return unallocated pages from the central queue to the thread requesting memory. We use CAS here to atomically move the head to the next node to delete the first node. The head node is never NULL since the algorithm returns without deleting any nodes when there is only one node in the list. Once the CAS operations are successful, the deleted node is assigned to one of the requesting threads.

Once we had our lock-free version for the central queue ready, we ran the program again on the same workloads as mentioned in Approach 2. The results obtained are explained below:

1. Workload1: VVMalloc does better than both custom malloc and ptmalloc

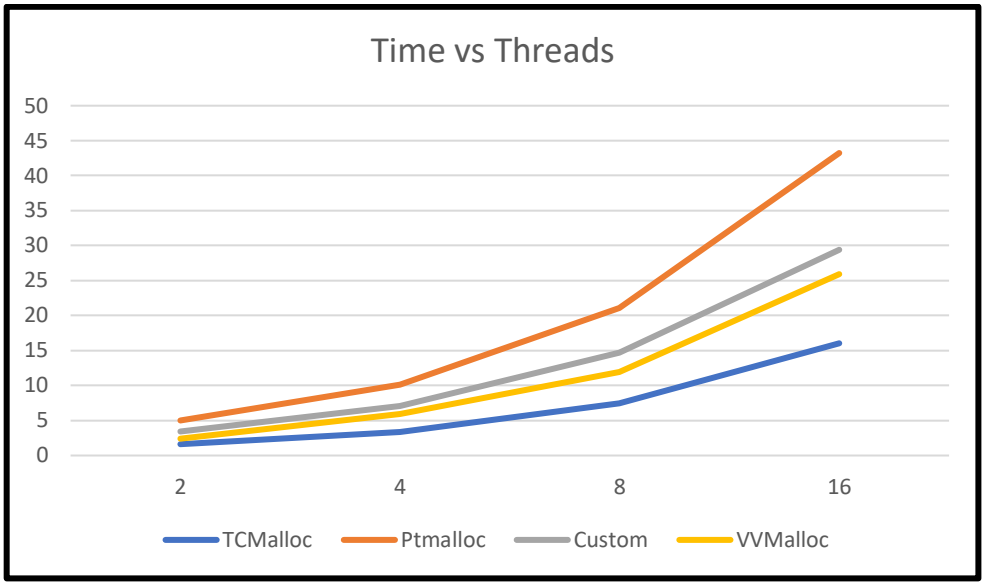


Fig 10: Approach 3, Workload 1 – Time taken vs number of threads

2. Workload 2: VVMalloc does worse than ptmalloc and pretty much overlaps with custom malloc

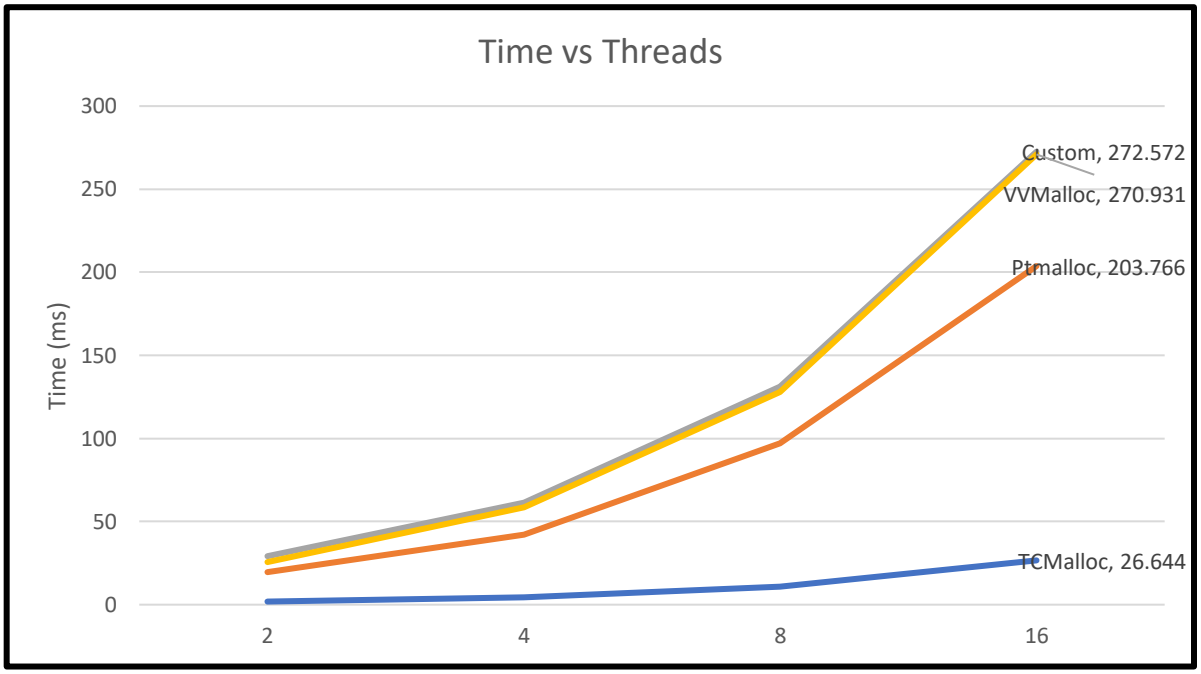


Fig 11: Approach 3, Workload 2 – Time taken vs number of threads

Workload 3: VVMalloc does better than ptmalloc but is comparable with custom malloc.

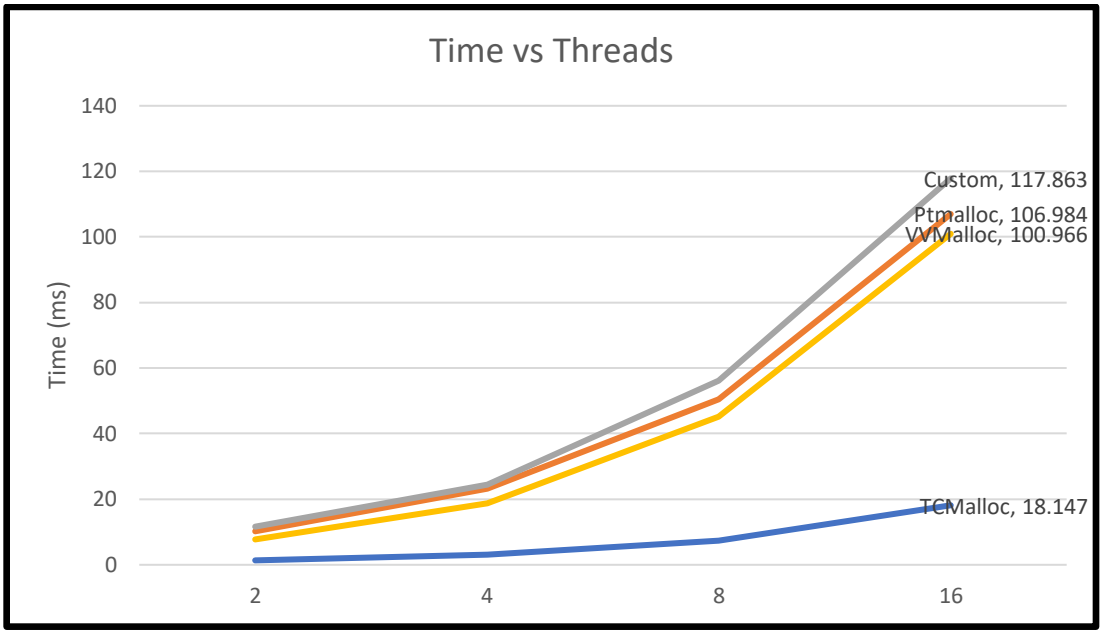


Fig 12: Approach 3, Workload 3 – Time taken vs number of threads

The graphs obtained above came as a surprise to us as we were expecting the lock free version of the queue to perform significantly better than custom malloc implementation in Approach 3. To understand what was happening, we further ran perf on Approach 3 and inferred the following:

1. The program is bound by the overhead due to spinlock (89% overhead according to perf) protecting the heap thus being the major bottleneck.
2. In order to significantly improve our performance from Approach 3, we needed to modify our algorithm such that the repeated access to the heap is further reduced.
3. One way to do this was to allot a large enough arena initially to the threads such that there is sufficient pre-assigned memory for majority of the requests to reduce contention on the heap

4. One other aspect we realized was that the central queue is highly contented with enqueue and dequeue operations causing an overhead of 4%. In our lock-free implementation, the overhead caused due to spinning is approximately the same as that caused by using mutex locks. This is evident for Workload 2 and 3 as the threads exhaust the assigned arenas frequently.

Based on our analysis above, we further modified our approach as explained in Approach 4 next.

Approach 4: VVMalloc - Improvised

To ensure that the access to the heap is reduced, one straightforward approach was to provide each thread sufficiently large arenas (the minimum size now being 256k) initially, so that corresponding requests would be allocated memory from the thread local storage and cause less contention on the global heap. As expected, this significantly improved the performance of the allocator. This directly translated to the execution time and is seen to perform way better than ptmalloc. To be sure, we ran perf to check the effects of increasing the arena size, and the overhead caused due to the spinlock came down. Since the metric is in terms of a percentage, the numbers shown may seem to be insignificant, but considering 92% of a bigger execution time is large as compared to 88% of a smaller number put things into perspective for us.

Workload 1

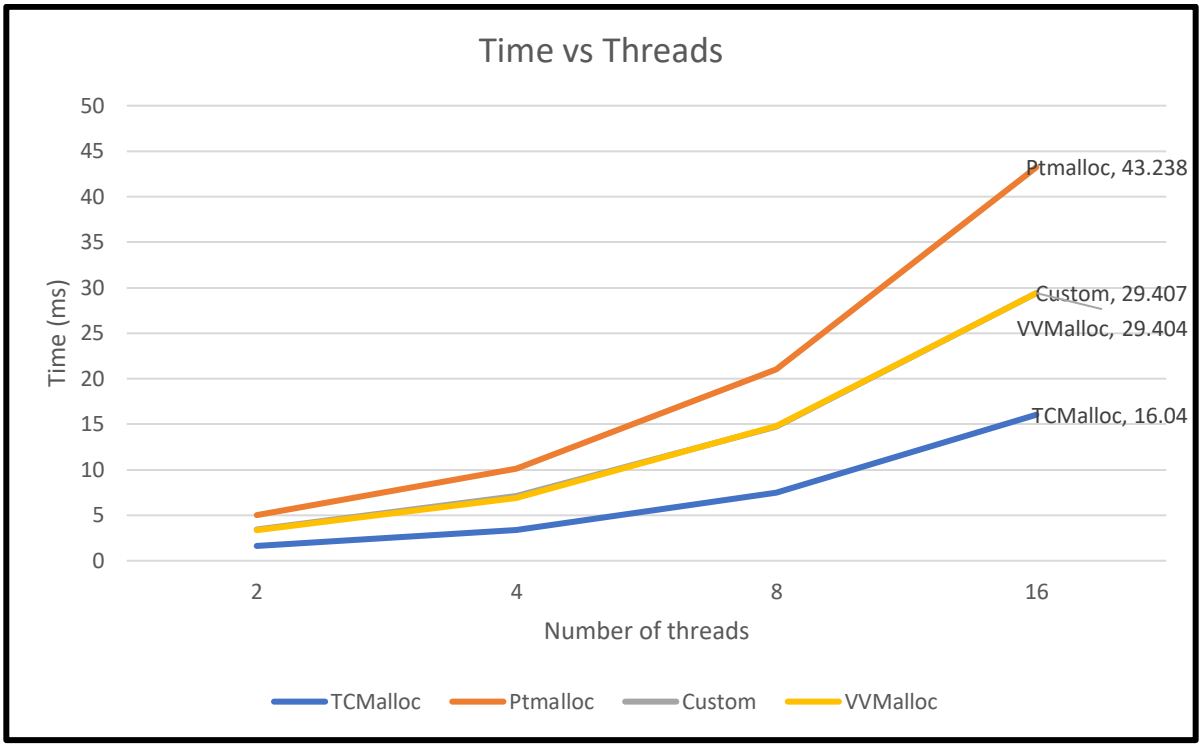


Fig 13: Approach 4, Workload 1 – Time taken vs number of threads

Workload 2

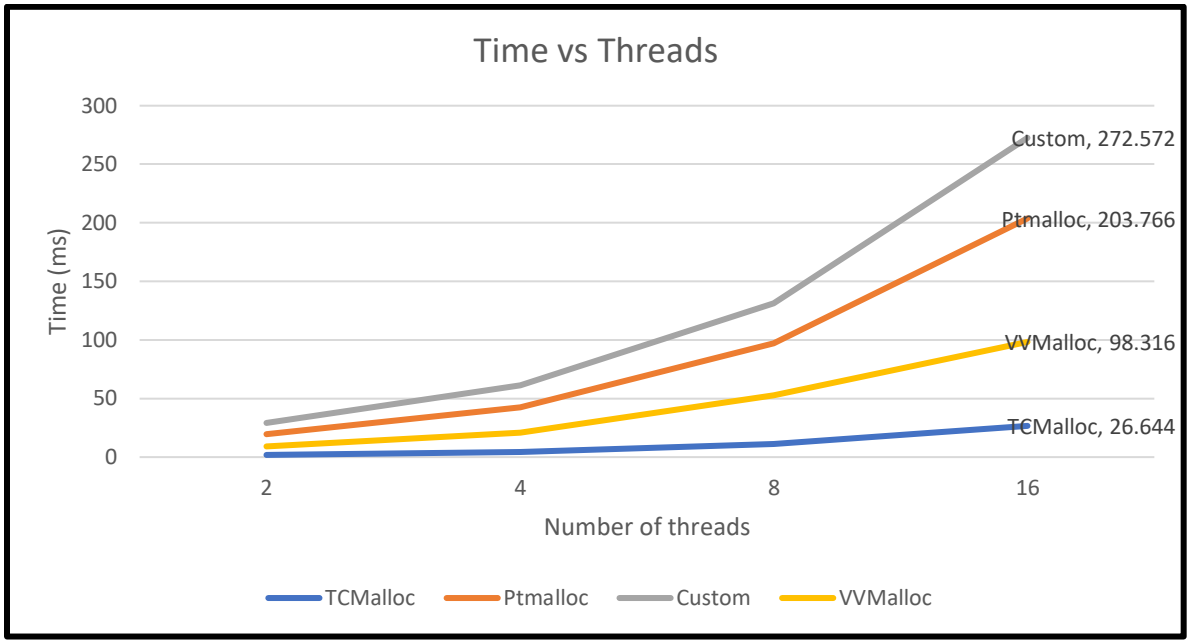


Fig 14: Approach 4, Workload 2 – Time taken vs number of threads

Workload 3

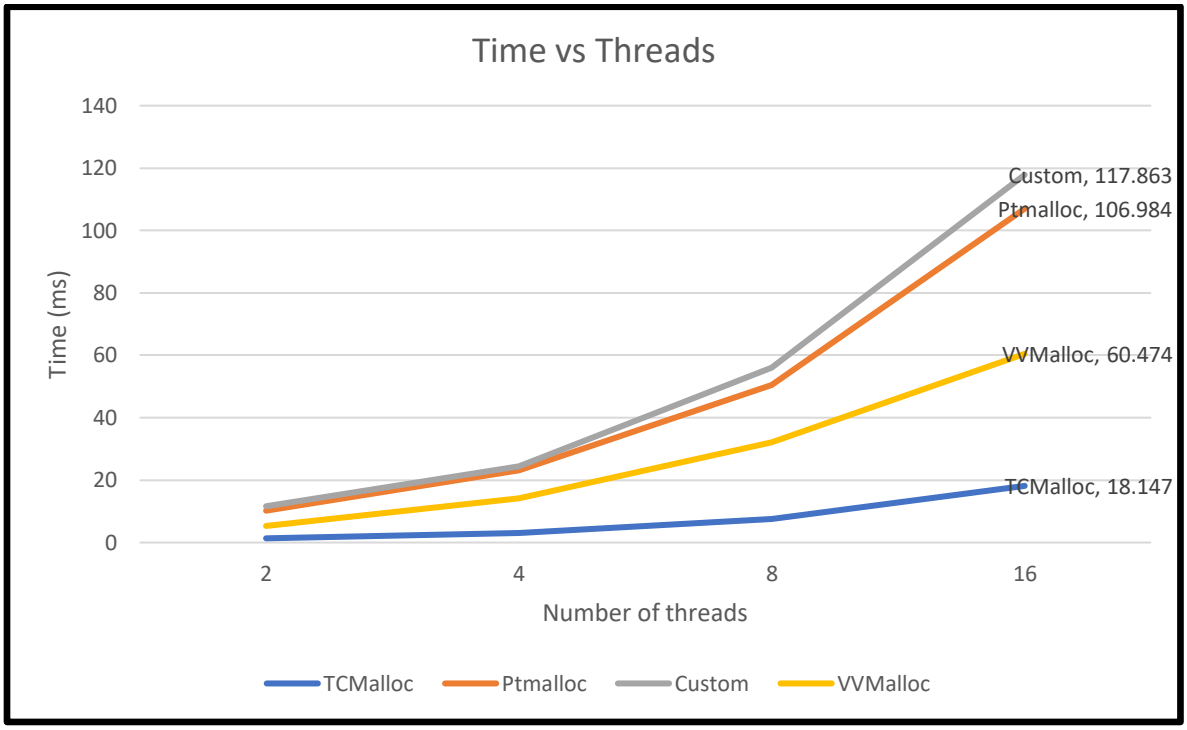


Fig 15: Approach 3, Workload 3 – Time taken vs number of threads

As per the timing plots above, we could draw the following conclusions:

1. For workload 1, we see that the timing plots overlap for VVMalloc and the custom malloc implementation. Since the allocation requests here are less than 32 bytes, the initial chunk of memory assigned to the thread is more than sufficient to handle the workload. Therefore, increasing the arena size has no effect in this case. The speedup in this case is barely over 1x as compared to custom malloc thus backing the theory regarding the small allocation requests.
2. For workload 2, the gap in timing between custom malloc and VVMalloc is far more pronounced. This is because workload 2 has its allocation requests which are in range of 3kb to 20kb. By providing a larger arena initially to a thread, this puts less contention on the global heap/ central queue since the thread is allowed to work on more requests before it can request for a new arena from the queue. This in turn ensures that the overhead due to spinlock reduces. The speedup in this case is ~3x consistently across different thread counts as compared to custom malloc which provides better understanding regarding the larger allocation requests in this workload. This also reduces contention on the central queue thus reducing overhead due to spinning till CAS succeeds
3. For workload 3, the gap in timing is also pronounced, although not as much as is present in workload 2. This workload consists of a mix of large and small allocation requests. Going by the same analysis, the larger arena size provided aids the bigger allocation requests and prevents frequent access to the central queue and in turn the global heap. This

provides a speedup of ~2x consistently and thus makes us believe that the arena size itself has a big role to play in the memory allocator.

RESULTS

The aim of the project was to implement a custom version of malloc – VVMalloc which could handle multiple threads, perform several operations in parallel all while maintaining correctness and with a good speedup as compared to the baseline single threaded implementation of memory allocator. Over the course of this project, we were able to successfully do what we aimed to by incrementally building the allocator and using techniques taught in class as well from multiple references. The ideal starting point which we thought was perfect was the 213 malloc lab code since it had the framework setup well enough on which we could tweak and modify heavily to test our implementation thoroughly.

From the perspective of testing, we were able to validate our results for every approach that we went with by making use of roughly 25 of the trace files present from the existing infrastructure. Some of these traces were especially helpful in terms of providing the workload as described in the approaches above. Some of these traces had large request sizes while some of these traces had just small size requests and others a mix of large and small requests. As for debugging the code, we would write short traces to test edge cases to break the allocator.

The graphs below show speedup of VVMalloc with respect to the custom malloc mentioned above as well as the ptmalloc which is provided as part of libc implementation for the 3 kinds of workloads mentioned in the approach section of the report.

1. Workload 1

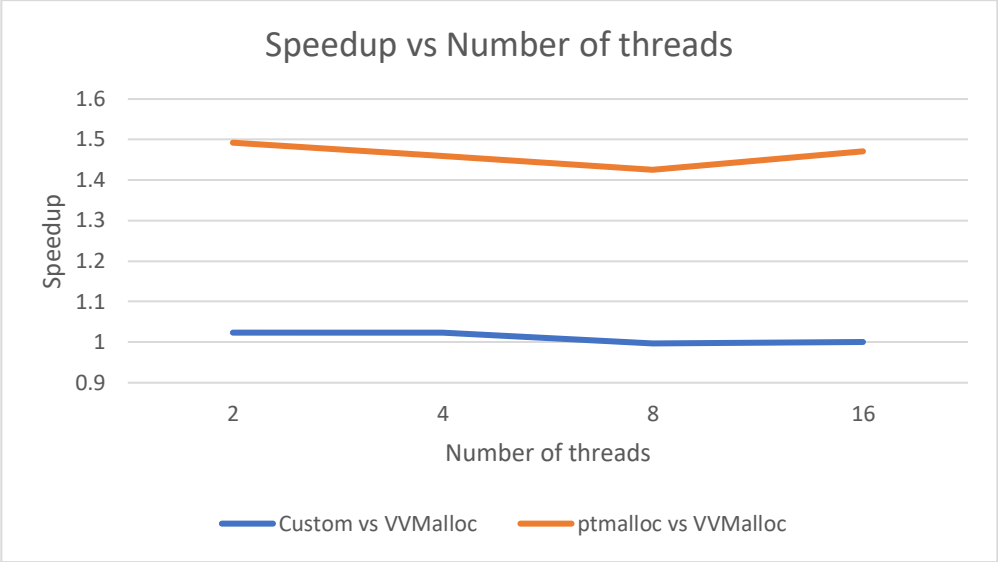


Fig 16: Workload 1 – timing speedup vs number of threads

With respect to the custom malloc implementation, VVMalloc performs quite similarly and hence we see no speedup. The reason for this being the small allocation requests do not benefit from the lock free implementation or the increased size of the arenas as the threads don't access the central queue or the heap as often for subsequent memory requests. On the other hand, we see a 1.5x speedup across different number of threads for VVMalloc with respect to ptmalloc.

2. Workload 2 – Large Size Requests

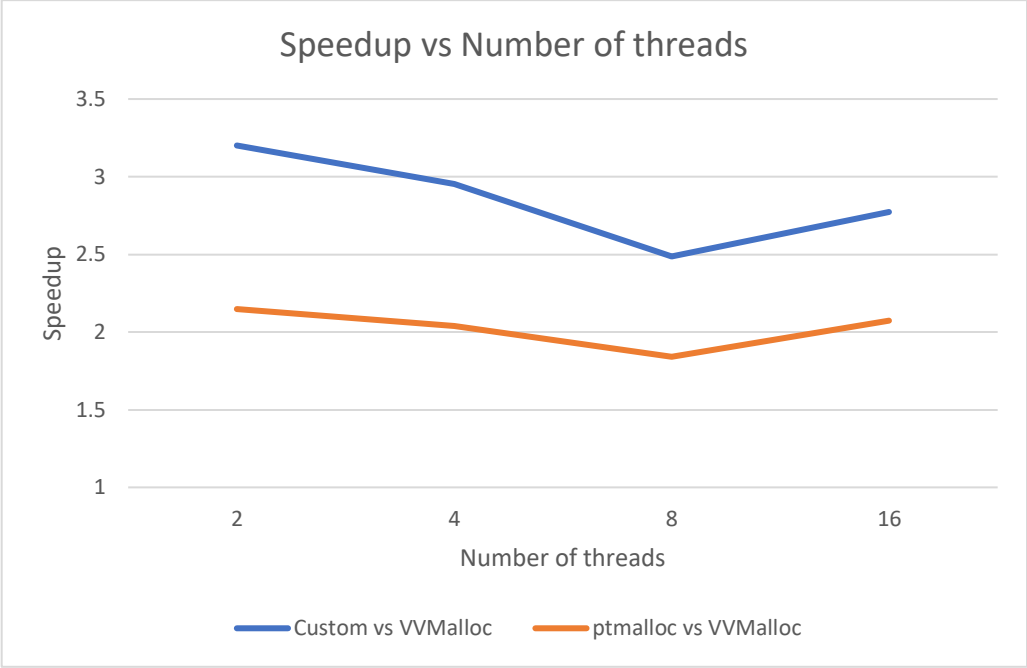


Fig 17: Workload 2 – Timing speedup vs number of threads

Here, we see how using VVMalloc can start to maximize speedup benefits. While comparing it to custom implementation, we get a huge 3x speedup on average across threads since the increased arena size assigned to thread local list ensures that it is not accessing the global structure frequently. Our implementation also performs twice as better as ptmalloc for workloads with large allocation requests.

3. Workload 3 – Mixed Size Requests

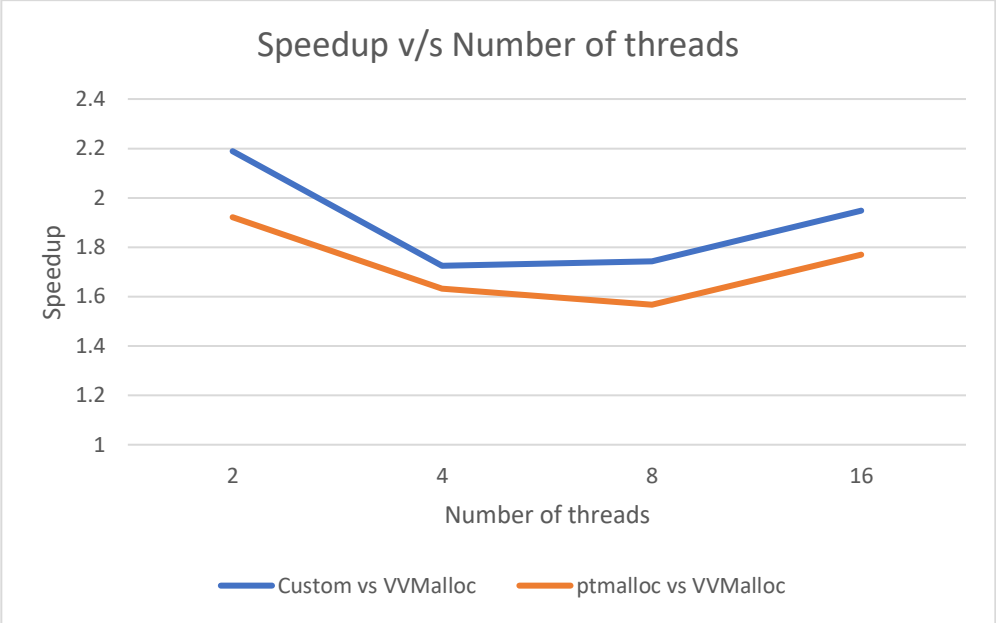


Fig 18: Workload 3 – speedup vs number of threads

For mixed size requests, the speedup is as expected and is around 2x as compared to the custom implementation. This comes because of having sparse requests for an arena from the central queue. Although, it is not as frequent as the workload consisting of large requests, there definitely is the access to the global structure from time to time. VVMalloc still performs better than ptmalloc across threads with speedup varying from close to 2x to consistently around 1.5x. Overall, VVMalloc implementation is seen to perform better for different kinds of workloads. Although this isn't as quick as TCMalloc which performs very well because of utilizing thread local cache, it does perform significantly better than the implementation of ptmalloc.

The limiting factor would be multiple threads having to use a global structure be it the heap or the central queue. This translates to having to use a locking mechanism while a thread accesses

this structure. The fact that there exists a spinlock cripples the system in terms of speedup if there are multiples accesses to it. One way which we thought was best, was to increase the arena size so that the frequent accesses to this heap and queue reduce and thus in turn reduce the time a thread spins waiting for its turn. This reduces the overhead and thus improves performance as seen from the results presented above.

One of the reasons TCMalloc is faster than VVMalloc is because TCMalloc uses thread local caches. It utilizes the thread local cache to satisfy memory requests less than 32kB with objects being moved from central heap into thread local cache as needed. When no free blocks are available, the threads requests memory from the thread local cache instead of the central heap. This creates a second layer of abstraction which we couldn't implement due to time constraints. In our case, we make use thread local storage for the list of thread local free blocks and request memory from either the central queue or the heap when a thread runs of out free blocks.

Final Presentation Video: https://youtu.be/mqXMaoqvUMM

Private github link: https://github.com/vhkashyap/15618ProjMalloc

REFERENCES

[1] https://people.cs.umass.edu/~emery/pubs/berger-asplos2000.pdf

[2] https://codearcana.com/posts/2012/05/11/analysis-of-a-parallel-memory-allocator.html

[3] https://github.com/jemalloc/jemalloc

[4] http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[5] https://www.cs.rochester.edu/~scott/papers/1996_PODC_queues.pdf

[6] https://github.com/nimsnaik/Lock-Free-Stack-Implementation-using-Custom-Parallel-Malloc

WORK DIVISION

Work Split: 50 – 50

Vignesh: Thread safe malloc, Custom Malloc

Vaishnavi: Lock Free Central Queue, Testing Infrastructure

Both: Project Proposal, Literature Survey, Performance Analysis, Final Report