15618 - Parallel Computer Architecture Project

Nimita Naik and Sharan Turlapati

Lock Free Stack Implementation using Parallel Malloc

SUMMARY:

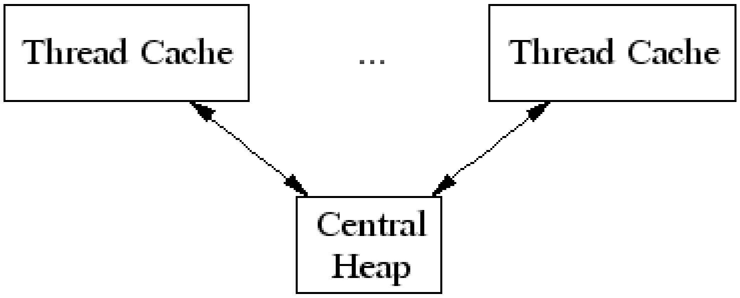
Implementation of a lock-free stack using linked lists that scales well when multiple threads are used adapted from “A Scalable Lock-free Stack Algorithm” by Nir Shavit et all running using a custom parallel malloc adapted from the implementation of TC Malloc.

BACKGROUND:

The implementations for multi-threaded parallel malloc we referred to are Intel’s tbbmalloc, jemalloc, tcmalloc, hoard and nedmalloc. A common theme across these implementations is to make a best attempt at servicing the request without locks failing which a central memory allocator is used. Based on the availability of details of the implementation and our estimate of the time needed to implement these, tcmalloc seemed most suitable for the purposes of this project.

TC Malloc Library Implementation

The TC Malloc assigns each thread its own thread local cache. Objects are moved from the central heap into the thread local cache and periodic garbage collection is used to move the memory back.



TC Malloc manages the heap as a span – which is a sequence of pages. A span is used to allocate memory to the central heap and the thread local caches.

Allocation:

TC Malloc distinguishes requests based on the size. Sizes below a certain threshold will be serviced by the Thread Cache while those greater will be serviced by the central heap. The span that is used to service the request will be marked as belonging to a larger object(serviced from the central heap) or a smaller object (serviced from the thread local cache).

Both the thread cache and central heap will maintain a free list of objects classified by the size. Based on the size of the request, the corresponding list will be used to service the request.

Deallocation:

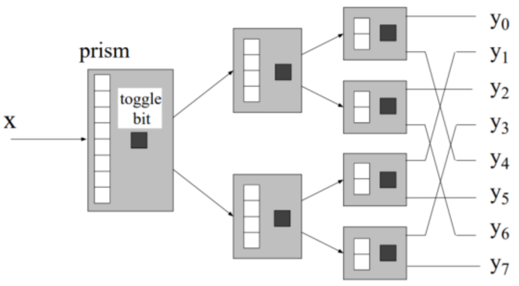
The span is used as a marker to figure out the list that the freed address belongs to. Based on the list returned, we will jump to either the thread cache list or the central heap.

Garbage Collection:

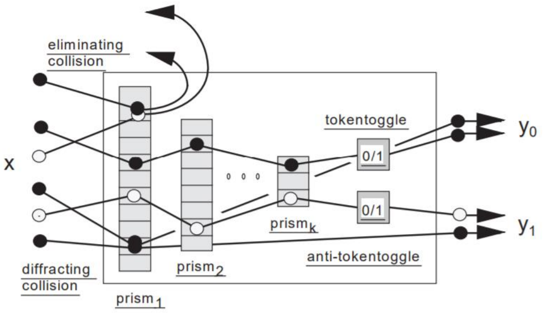
A thread cache is garbage collected if its object size exceeds a certain limit. This feature was done away with by making thread local caches have fixed sizes that are reused whenever possible.

Lock-free linked list stacks:

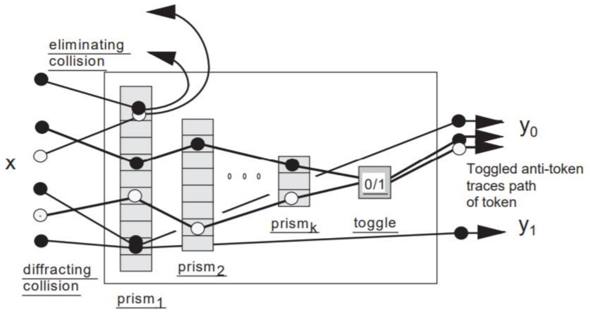
The first two papers reviewed were the diffracting tree and the elimination stack. While these are not optimal methods of creating a lock free linked list, understanding the logic behind these classic approaches was essential before implementing a lock free linked list. Diffracting trees make contention happen at various different points on a combining tree such that the final counter or value is incremented only by one thread whenever a write collision occurs, and the answer is then ‘diffracted’ out. The tree reduces contention between threads by ensuring that all the threads are not contending for a single shared variable. However, when this approach is used on a single threaded program, there is a very high overhead. In the sense it does not scale down well.



The elimination stack approach is slightly better in that there are multiple small trees dynamically created instead of a single large tree. While the overhead is lower than the diffracting tree, the algorithmic complexity is very high for not as much value-add in a simple stack implementation and the speed up achieved is not significant.

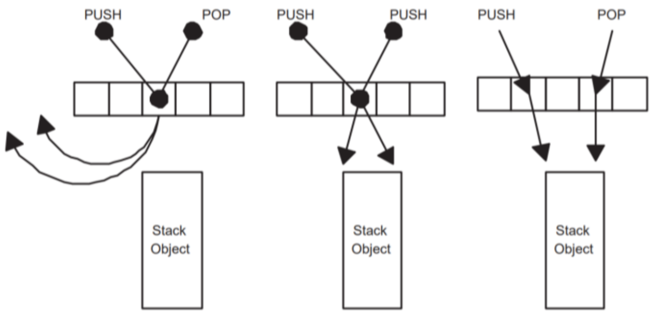


Pool collision balancer

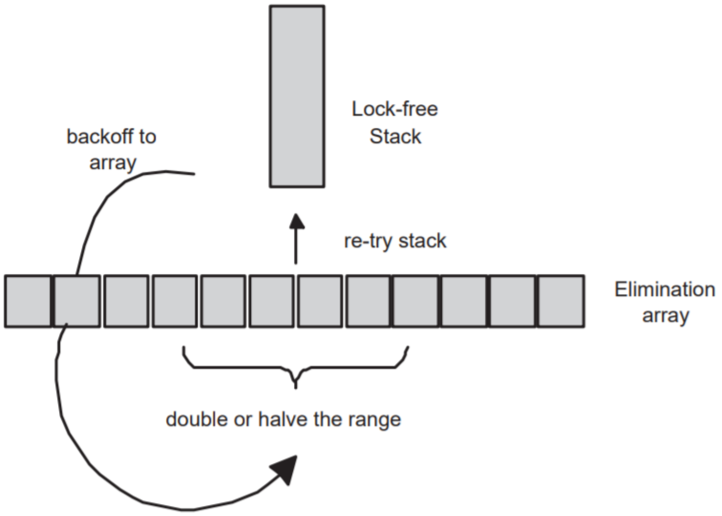


Stack collision balancer

The paper we are implementing uses elimination back-off array, where a thread first tries to put its value on the shared stack, if it faces contention, it then goes to a collision array where collisions are handled. Push/Pop collisions are handled by replacing the head with the new head that is to be pushed. If there is no collision or a “like” (push/push) collision then the thread, after waiting for a fixed period tries to add directly to the stack again. Unlike the funnel and tree approach, this method has no overhead when no contentions are present.



In the above image, we can see that a push-pop collision is dealt with without going to the main stack at all. A collision array is maintained to manage these collisions and threads alternatively try their operations on the collision array and the main stack with varying back off times until they are successful as shown in the figure below.



MACHINE USED:

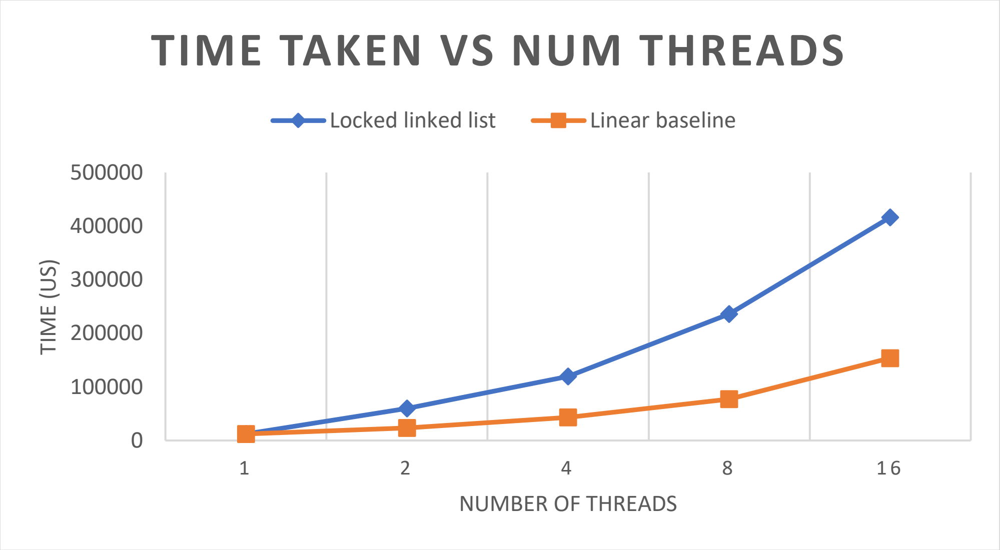
The project was tested on the GHC machines which have 8 cores with a 16-way hyper-threading. Some initial malloc tests were performed on the shark machines, which has the same core count but a slightly lower clock rate. All tests measurements in a single graph are taken consistently on the same machine. The machine limitation is what prompted us to take measurements only up to 16 threads. We were happy with our choice of machine.

APPROACH:

Part I: Lock-free Linked-list Stack:

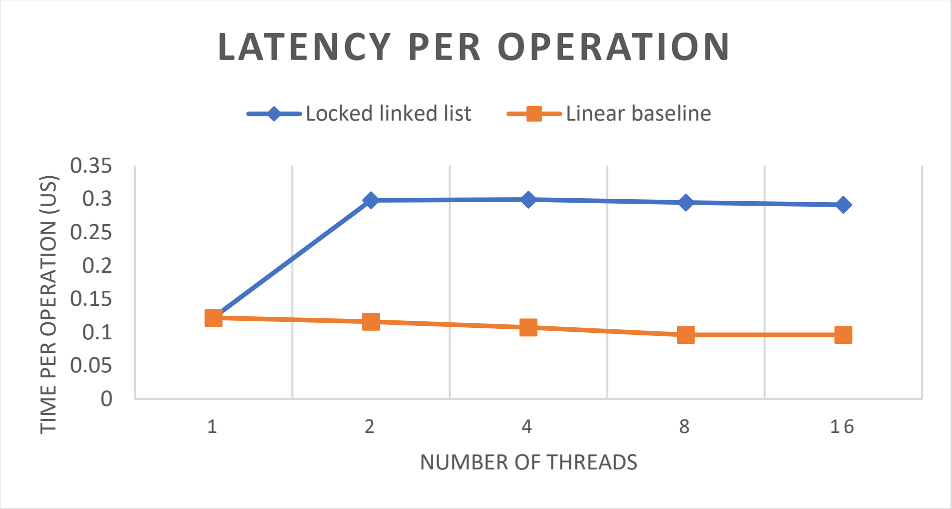
1. Locked Stack Using Linked List

The first method was a simple lock free linked list using a mutex. This was our baseline implementation that we hoped to speedup using lock free stacks. There was a single mutex available to all the threads and each thread would lock before accessing the stack. The access pattern used is pseudo-random. While we do test with other patterns, we do not want to optimize to a specific type of list access pattern. The speed we get using this approach is as follows.



In our simulated workload, each thread performs 100000 operations. As the number of threads performing these operations increases, there is more contention between the threads. The linear baseline is the time taken to perform the same number of operations on a stack in a single threaded program. This line is our target. As the contention between threads increases, we want the time taken to be the same as performing all these operations serially, making the cost of contention 0.

We can clearly see in the graph that the simple locked version performs poorly as the number of threads increases. The mutex is highly contended by all of the threads and causes a huge amount of latency per operation.



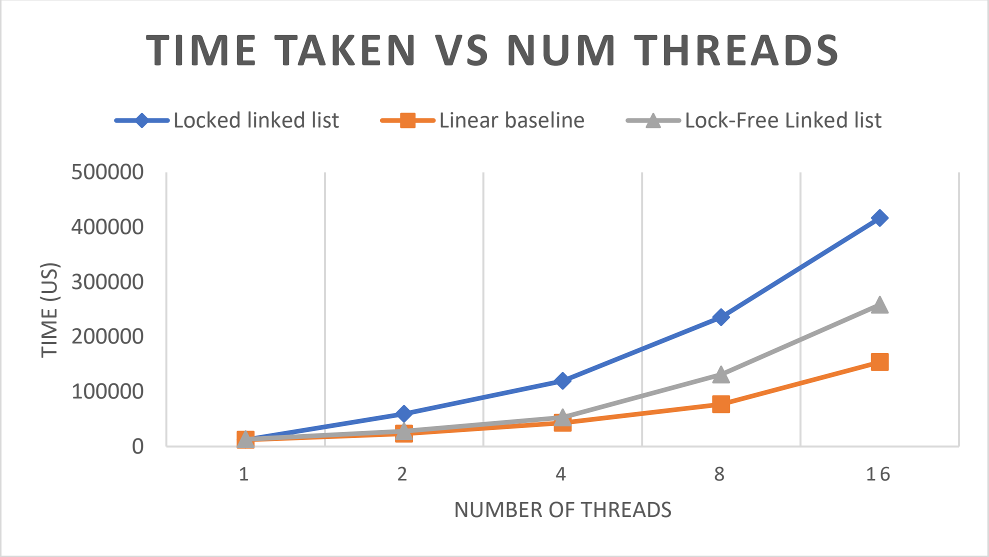
This graph shows that the time taken per operation increases almost 3-fold when there is contention between threads. We considered implementing a fine-grained locking technique in order to reduce the latency. However, a stack insertion and deletion always happens at the same point (the top of the stack) and thus fine-grained locking would not be a good idea.

Thus, we use this locked code as the baseline we want to improve on and the linear baseline as our target. For all the following linked lists, we maintain the same number of operations and the same pseudo-random pattern.

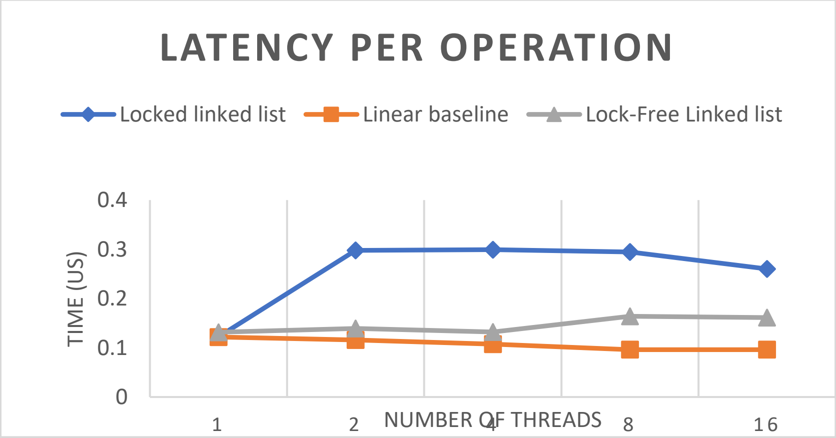
2. Lock-free linked list

Next, we implemented a lock-free version of the same linked list. In this version we use the compare-and-swap operation to insert or remove a node from the top of the stack.

A challenge we faced was freeing allocated blocks. Since other threads could be using CAS on the pointers, they could not be freed. We implemented a 'hazard pointer' mechanism. Each thread has a pre-assigned list of hazard pointers which it can fill. When a pointer must be freed, it is added to the 'free list'. When the free list reaches a certain size, we go through the entire list and free the pointers which are not pointed to by the hazard pointer of any list. Since only the thread itself can write to the hazard pointers in its own list, there is no contention. There is the possibility that the free list is full but none of the objects can be freed, but we overcome this scenario by making the free list size larger than the total number of hazard pointers that all threads can hold.



As we can see the lock free implementation performs very well when there are a low number of threads. However, when the contention increases, there is some delay experienced. Additionally, since the freeing is done via hazard pointers, it does add some overhead to the process even when only 1 or 2 threads are present.

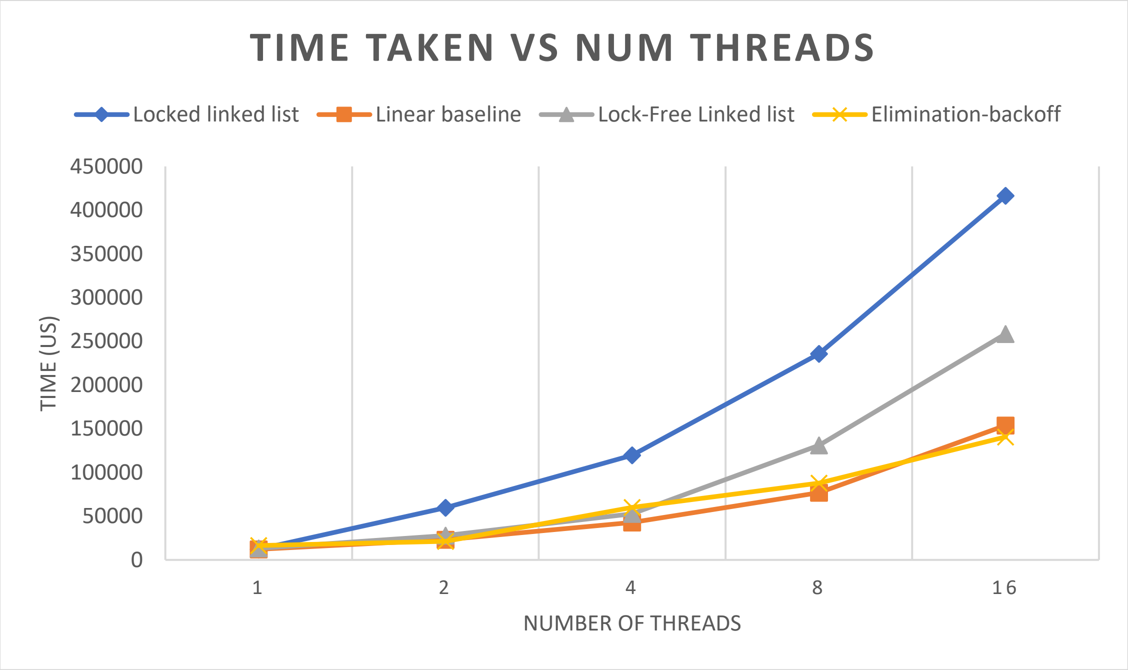


In this graph as well, we can see that although the lock free is close to having no overhead when there are fewer threads, there is a lot of contention when the number of threads increase.

3. Elimination back-off based linked list

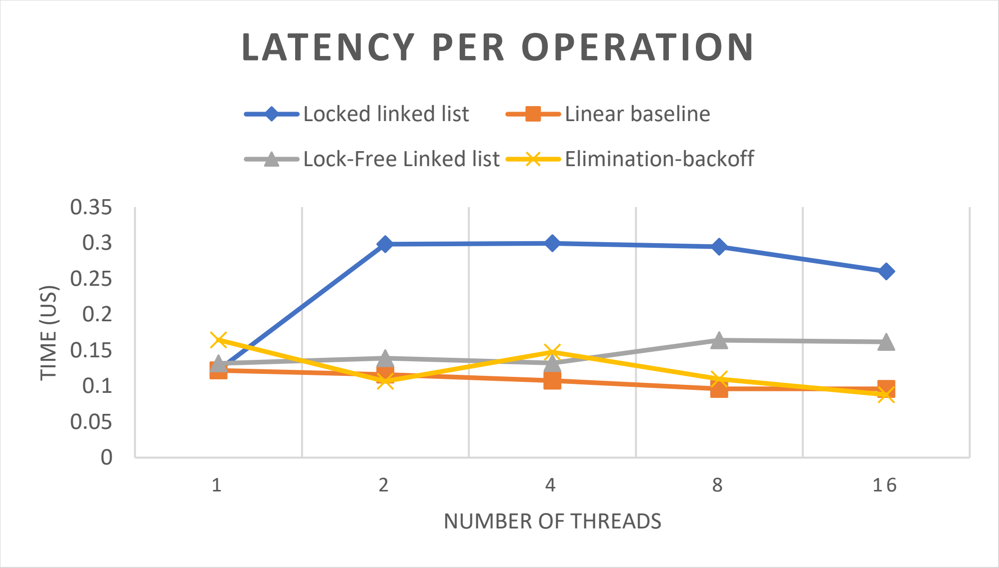
As we sought to improve the performance to at-least the linear baseline we decided to implement the elimination based back-off stack as given in the paper "A Scalable Lock-free Stack Algorithm" by Nir Shavit et all. In this method, a 'collision array' is maintained for each thread. Once a thread fails to perform the main operation on the main stack, it goes to the collision array. If the thread encounters another thread which is doing the opposite operation (PUSH/POP) on the stack, then they exchange the node amongst themselves without going to the stack at all. If a thread is not collided with successfully on the collision array, it retries on the main stack and the cycle continues until the operation is successful. The thread also varies the amount of wait time it spends on the collision array as a 'back off' parameter.

Once we had the working implementation of the elimination back off method with no freeing, we faced a significant challenge refactoring the code to support freeing of cells using Hazard Pointers.



In this graph, we observe that the elimination backoff performs as well as the linear baseline even when we have multiple threads. In fact, it outperforms the linear version because pushes and pops collide with each other without going to the main stack at all. In the above graph, we optimized the backoff parameters and the collision array position of each element in order to give the best performance. As the access pattern is random, we believe this is a good general optimization.

Since all the threads access the collision array, we found that adding some padding within the array made the program faster due to cache access patterns.



We observe that the average latency for operation in elimination backoff is approximately the same as our linear baseline and it scales well.

Testing:

A challenge we faced was successfully testing each list implementation to ensure that it was correct. The following tests were performed:

- run the program 100 times with over a million push/pops to ensure that it never fails
- count the number of pushes and successful pops (non-NULL return) and ensure that it tallies with the number of elements left in the list at the end of execution

- count the number of frees performed and the number of items left in the hazard pointer list and ensure that they match up

Although non-exhaustive, we believe that these tests confirm to a reasonable degree that our implementation is correct.

Part II: Our Implementation of Parallel Malloc:

Baseline implementation

Our initial few days was spent on setting up a baseline that we can use to measure speed up for our implementation of parallel malloc. We decided to re-use our 15-213 malloc implementation and make it thread safe. The current malloc implementation was modified to acquire and release a single global lock upon every malloc and free request. To test the working, we decided to also reuse the trace files that the lab uses. The current mdriver implementation of the lab runs each trace file one after the other, restoring the heap to a clean state every time before starting a new test.

We modified the mdriver code to run all the traces in parallel. We used pthreads to spawn as many threads as there are traces and assigned one trace to each thread. The heap will be initialized to a clean state once and will continue to receive requests until all threads have finished running their assigned trace file. While doing so, we retained the correctness checks, utilization and throughput calculations carried out by the mdriver code. Each thread runs each of these checks with the trace that it is assigned to.

Adapting the mdriver code structure, especially the timing measurement logic to work with this design did take up a few days but we were finally able to get it up and running.

Below is the result of our serial malloc implementation and thread safe parallel malloc implementation running on a shark machine measured in terms of average throughput obtained –

Serial Malloc – 17795 Kops/s

Thread safe parallel malloc (baseline) – 3324 Kops/s

We will be using the above thread safe malloc as our baseline for speed up measurements.

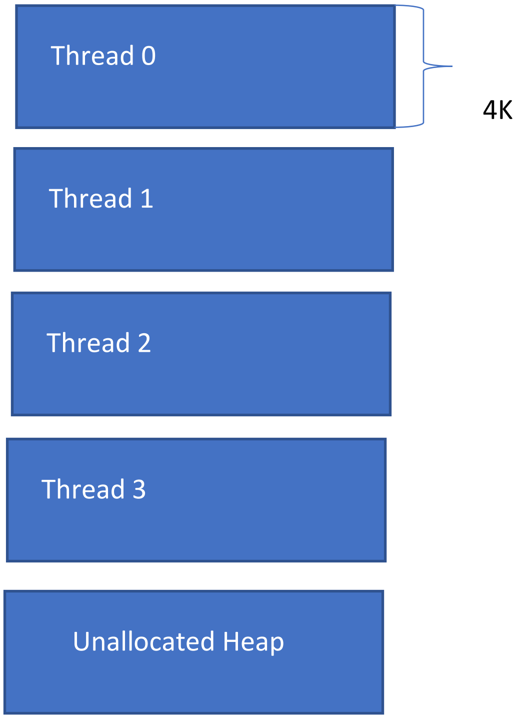
<u>Optimized Malloc Functions:</u>

1. Initialization

Upon boot up, a segregated list is created for each thread cache. Each thread cache is associated with a unique ID that starts from 0 and spans up to (number of thread caches created).

A block of heap of size 'chunksize' from the central heap is allocated to each thread cache. After each allocation to a thread, the thread ID field in the block is populated with the thread ID of the thread that it is allocated to.

The thread cache code will add it to the appropriate entry in its segregated list based on the size. The heap may look like this for 4 thread caches and a chunksize of 4K after initialization where the thread ID indicates the thread cache ID that the particular region of the heap belongs –



2. Malloc

The code maintains a global variable “producer” that keeps track of the last thread cache that was used to service a request. This is initialized to zero. The producer is a modulo counter that increments by 1 with each malloc requests and wraps around when the number of requests exceeds the number of thread caches created.

When a malloc request arrives, the producer global variable is atomically read into a local variable and its value is atomically incremented by 1. The value saved in the local variable is the index of the thread cache that will be used to service the given malloc request. Since each thread cache is associated with a lock, the index value will be used to acquire the necessary lock. This ensures that the chosen thread cache will not be used to service any other request while this is in progress. The next request that comes in however will not be hindered as long as a thread cache is free to service a request.

The logic implemented to choose which thread cache to use for a service request is thus round robin. The thread cache index that runs for a given request will be the next available thread cache from the index used for the last request.

Once a thread cache is chosen, it will iterate through its free list starting from block size that greater than or equal to the request size rounded up to maintain meta data and alignment requirements. The thread cache uses a “good enough fit” allocation policy. If a block size greater than the size requested is used, it is split to form a smaller block (subject to metadata constraints) and is added to the appropriate free list entry. If a thread cache is unable to find a free entry in any of its entries, it will atomically request memory from the central heap. The central heap always returns ‘chunksize’ blocks. Therefore, a chunksize block is added to this thread cache’s free list. This also means that a thread cache does not pass the baton to another thread cache if it is unable to service a request with its currently available resources.

The chosen block is marked allocated, the succeeding block in the heap is made aware of this status before being returned to the user.

The round robin policy was chosen to ensure every thread cache gets a chance to run. Since the thread caches do not hand off requests between one another, this was a way to ensure fairness. It also ensures that the chunksizes allocated during initialization all get used and heap memory does not go to waste.

3. **Free**

Once a free is called with the appropriate address, the appropriate error checks are performed on the address. The payload address is then used to jump to the block's header. The block is marked as free and the succeeding block in the heap is made aware of this. The thread_id block is then used to add the block to the appropriate block's free list. However before adding the block to the free list, an attempt to perform coalescing is made. The coalescing is done only with neighbouring blocks that belong to the same thread_id. If a neighbouring block is free but belongs to a different thread_id, it will not be coalesced with this given block.

The design to coalesce between blocks of only the same ID has a few pros and cons. This will lead to potentially small segments of successive free blocks in the heap. However since they belong to the free list of different thread IDs, they may be reused from their respective free lists to service their own requests. This may or may not bring down utilization. We performed an experiment to indeed to see if this was a good design choice and is highlighted in the performance analysis section.

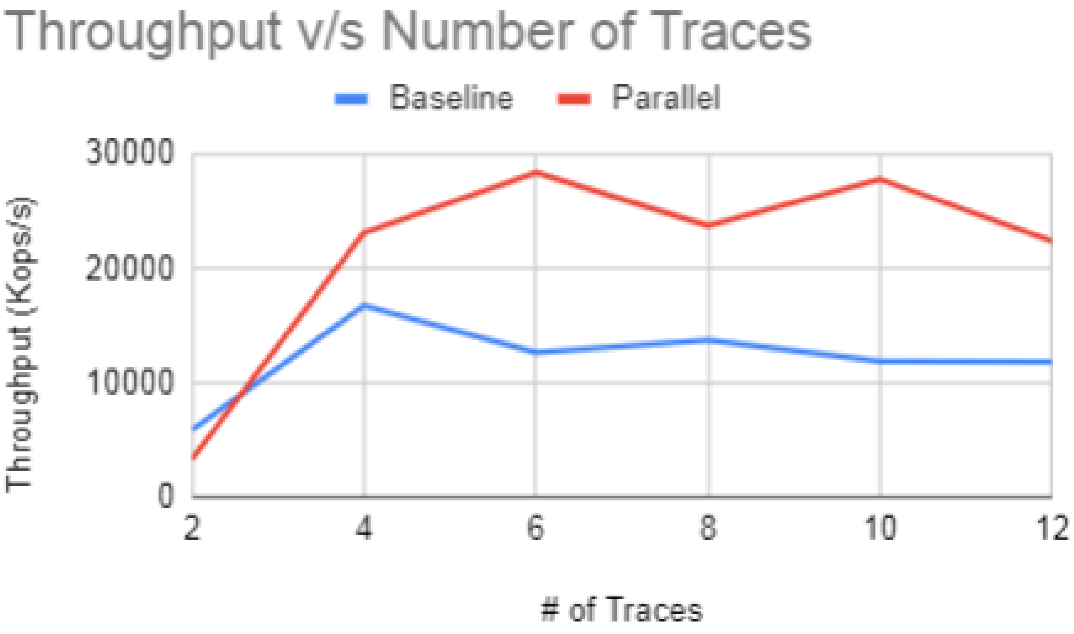
Performance analysis

For the performance analysis, the following pre conditions are defined. We used 12 traces from the 15-213 malloc implementation and the mdriver code was modified as described in the baseline implementation.

The performance analysis was made on the shark machines on an Intel Xeon. We were mostly interested in measuring the performance of the two lock designs. We expect any synchronization/communication overheads to be uniform across both design and thus nullifying their effects. Traces with a very small number (<100) of malloc/free requests were not used in order to remove any false spike ups contributing to average throughout. The mdriver code was modified to measure the throughput for each trace in parallel and the average of the throughputs of every trace run was used.

Baseline thread-safe implementation v/s Parallel Malloc

The following graph plots the throughput v/s no of traces performance for the baseline and parallel implementation. More number of traces essentially implies a greater number of requests in parallel.



Initially with just 2 traces running in parallel, the several lock acquisitions and releases in the parallel malloc end up hindering the performance and the baseline outperforms the parallel implementation. But as we introduce more and more requests in parallel, the parallel implementation takes off and outperforms baseline by a comfortable margin.

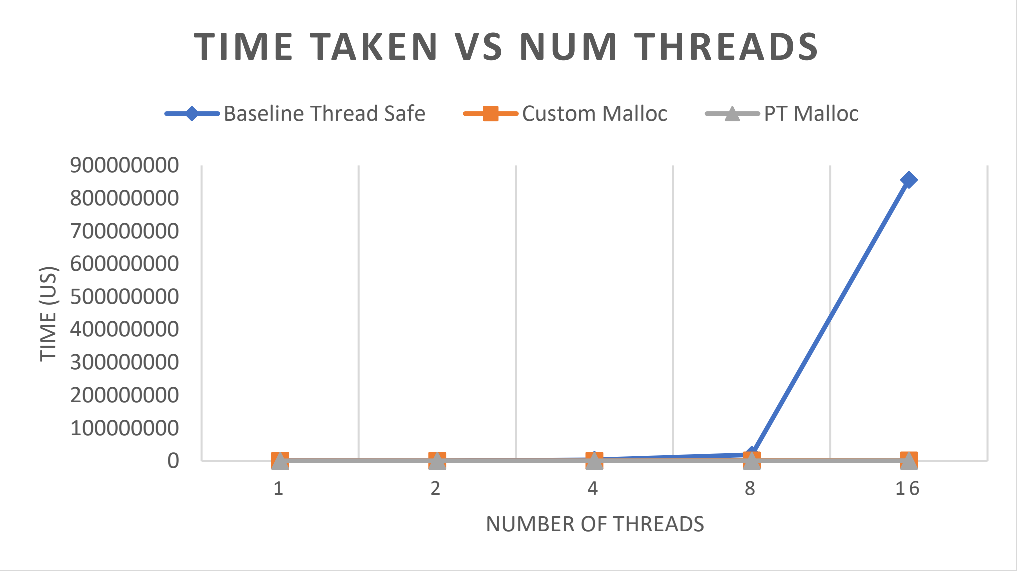
With increasing parallel requests, the contention over a single resource (lock for the central heap) causes the baseline to take more and more time.

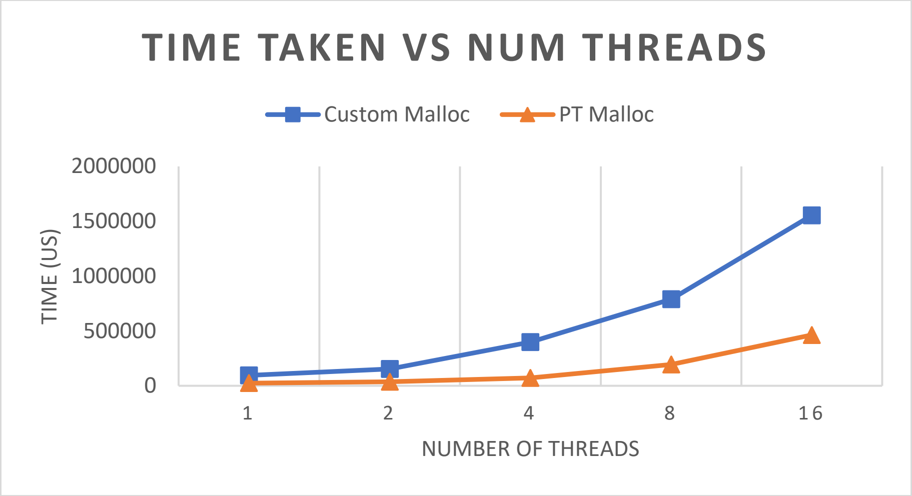
Even though the parallel implementation has several locks, the contention over a given lock is far lesser (lock for a thread cache) and thus the finer grain locking with lower contention proves to be effective in handling the parallel requests.

This was the design of the parallel malloc upon the top of which we performed integrated testing using the lock free structured code.

Part III: Elimination Back-Off Stack Linked List Stack with Parallel Malloc:

As our goal was to make a optimized stack and malloc implementation for a Parallel Computer, we used our custom Parallel Malloc along with the elimination back-off stack linked list stack implementation. Although we achieved a significant speedup in comparison with the original thread safe version, we were still falling short of the glibc malloc implementation. Something we had not foreseen was that when a program is compiled with -pthread, a malloc optimization called ‘PTMalloc’ which is already optimized for multithreaded programming using arenas is used. In our original proposal we had (incorrectly) assumed that glibc malloc was not optimized for multi-threading. Additionally, tc malloc shows great improvement over PTMalloc when multiple requests of varied sizes are involved, tc malloc is not optimized for multiple requests of the same size.





Despite this set back, we were determined to beat PT Malloc and brainstormed to come up with further ideas which we could implement.

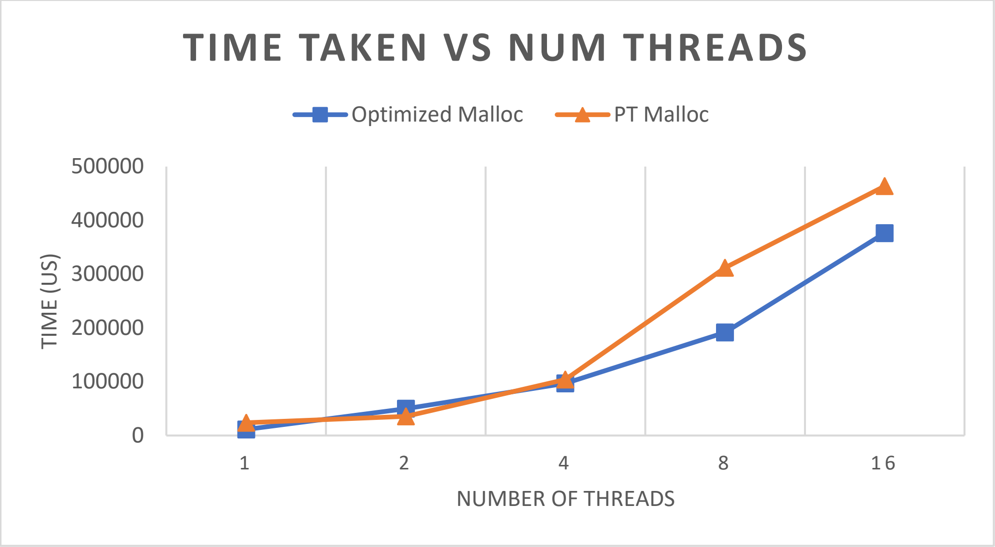
Ideas -

1) One major thing that we realized (this was mentioned in PCA lecture but the impact of it truly hit home when we saw it in our project) is that we were not compiling with -O3. Our optimization flag was still the default -O1. Once we compiled our malloc with an -O3 flag, we observed a significant improvement over the original implementation.

2) We stopped coalescing as all the malloc requests were of the same size in a linked list, coalescing added minimum value for this use case and cost a lot of time.

3) We also removed one lock (for round robin access of the thread caches) and used a counter that was atomically incremented using lock-free techniques (specifically we used the compare and swap operation instead of a mutex). This also gave us a significant improvement.

After performing these optimizations, we compared our implementation with PTMalloc and observed some speedup:

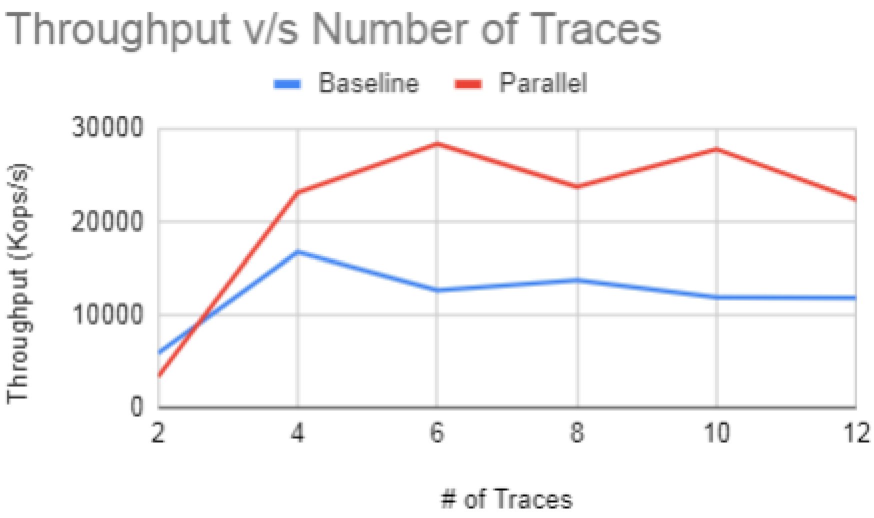


Here we observe a speedup in performance when compared with glibc malloc. Even on a single threaded implementation, there is a 2x speedup (not apparent in the graph as the values are so low).

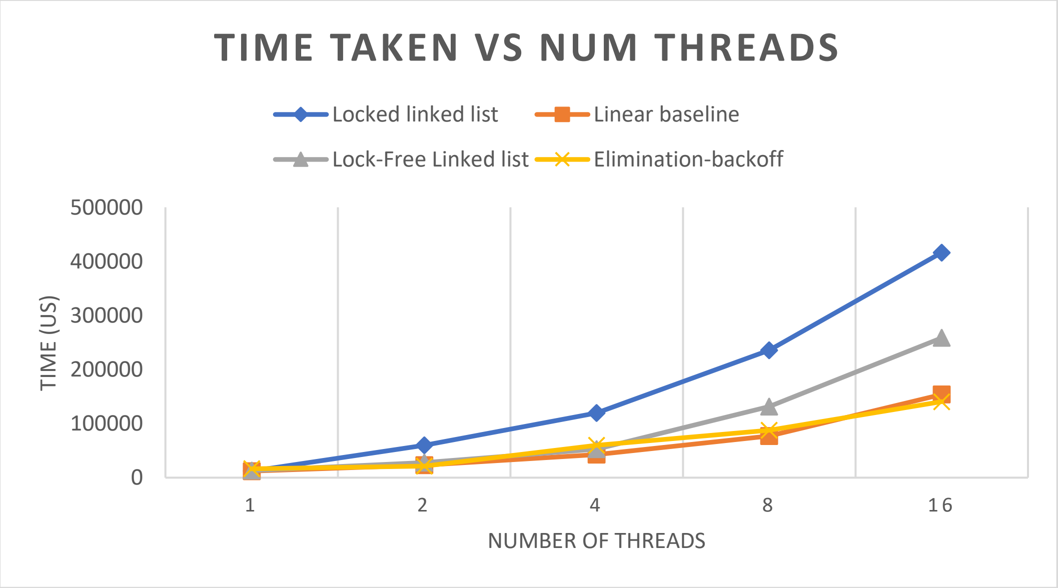
Another optimization that we tried is changing how free is implemented. Rather than re adding the free block back to its original thread cache, it was added in a round robin fashion to any thread cache. We believed that this would speed up the process as the malloc and free calls made by different threads would no longer compete for the same thread-cache lock. Although this was true, this optimization made the code slow down significantly. Since free blocks were returned to random thread caches, some got large while others had to keep increasing their size by expensive calls to extend heap. We did not see any specific access pattern being repeated or a cache optimization we could make apart from the above as there was no way to predict in what order the threads would push and pop onto the stack.

RESULTS AND CONCLUSION:

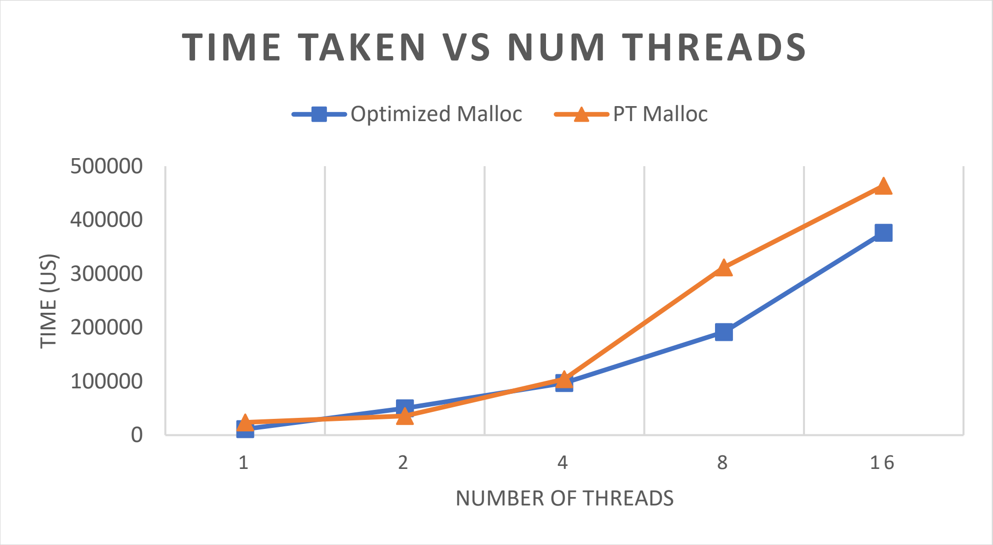
We implemented a parallel malloc successfully from a generic single threaded malloc that has the following speedup for traces of multiple varying request sizes :



We implemented a lock-free linked list using the elimination back-off method that took advantage of the collisions and used them to provide a slight speedup rather than the slow down due to contention that is observed both in generic locked and lock free techniques.

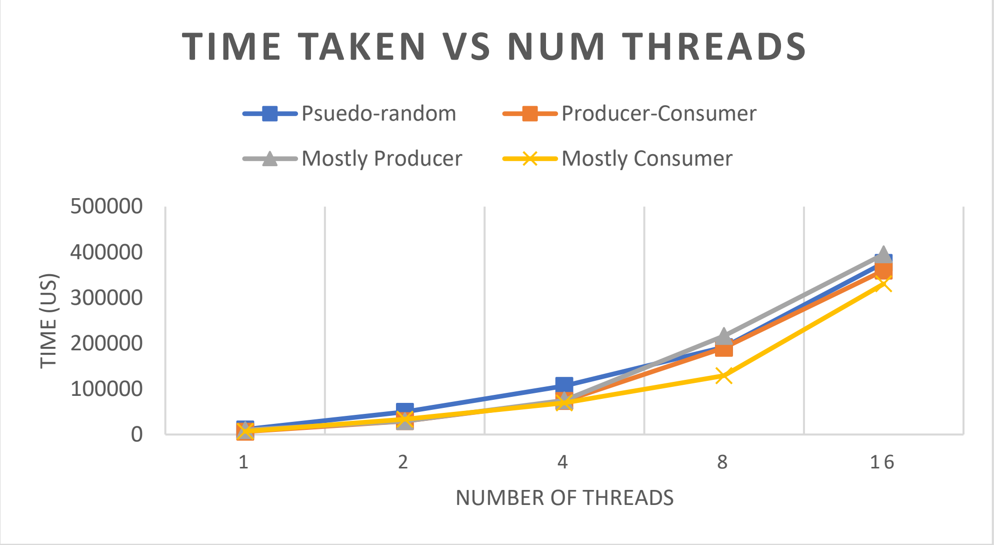


We also compared our implementation with glibc's PTMalloc and observed comparable performance with some speedup. The speedup was slightly lower than we expected because PTMalloc optimizes using arenas like what we had implemented and similar fine-grained locking. The main reasons we were able to observe a speed up are because we have a fixed number of thread caches optimized to the number of threads running in the program while PTMalloc uses a dynamic number of arenas which can increase to too many in case of 16 threads and reduce performance. Additionally, we do not coalesce free blocks to save time.



The main reason we are not much faster (as compared to say TCMalloc) is because we only have one layer of thread caches accessing the central heap. TCMalloc uses a second layer of abstraction, but this would involve implementing garbage collection of free blocks which we did not have the time to do.

We also observed the performance of our malloc for different types of access patterns. 1. Pseudo-random 2. Producer-Consumer 3. Mostly Producer 4. Mostly Consumer



In this graph we analyze various access patterns on our malloc implementation. We can see that the malloc implementation scales well to accommodate for the various access patterns. The difference in speed can be accounted for by the amount of memory required for each kind of thread – the ones which require less memory perform better. Also, from this graph it can be concluded that our lock-free stack implementation performs well for all access patterns.

The specific kind of code used for each of the above tests is given in further detail in the implementation section above.

REFERENCES:

Parallel Malloc:

http://www.cs.umass.edu/~emery/pubs/berger-asplos2000.pdf
http://goog-perftools.sourceforge.net/doc/tcmalloc.html
http://jemalloc.net/
http://codearcana.com/posts/2012/05/11/analysis-of-a-parallel-memory-allocator.html
https://www.nedprod.com/programs/portable/nedmalloc/

Lock-Free Linked-list Stack:

http://groups.csail.mit.edu/tds/papers/Shavit/SZ-diffracting.pdf
https://groups.csail.mit.edu/tds/papers/Shavit/ST-elimination.pdf
https://people.csail.mit.edu/shanir/publications/Lock_Free.pdf
https://github.com/azu-labs/Hazard-Pointers

WORK-DIVISION:

Grade Split – 50-50

Sharan: Thread-safe Malloc, Parallel Malloc

Nimita: Project Proposal, Lock-free and Lock-free elimination back off stacks

Both: Integration Testing and Optimization, Final Report