PCP(CS5300) Project Report

Team: P Sai Srikar - CS18BTECH11034
G V Sathwik Reddy - CS18BTECH11014
J Sai Nishith - CS18BTECH11018

We have implemented three variants of non-blocking queue based on the following papers:

MSQueue :

[1996] Maged Michael, Michael Scott "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms"

Optimistic Queue:

[2008] Edya Ladan-Mozes, Nir Shavit "An Optimistic Approach to Lock-Free FIFO Queues"

Basket Queue :

Source : [2007] Moshe Hoffman, Ori Shalev, Nir Shavit "The Baskets Queue"

We mentioned that we would implement RWQueue in place of which we implemented the MSQueue because RWQueue is a simple lock based queue.

MSQueue:

This is a linked list based implementation of FIFO queue. In this implementation of the queue any number of threads can do enqueue operations and any number of threads can do dequeue operations simultaneously. The following classes are used for the implementation of the queue.

```
class pointer_t
{
    public:
    node_t *ptr;
    int count;
}
```

```
class node_t
{
    public:
    int value;
    atomic<pointer_t> next;
}
```

```
class queue_t
{
    public:
    atomic<pointer_t> head;
    atomic<pointer_t>  tail;
}
```

The queue is a linked list of **pointer_t** objects.

When a thread comes to enqueue, it creates a new node_t object, initialize it with the value it wants to insert, then creates a new poiner_t object and initializes it. It then tries to add the node after the tail of the list and set itself to tail. To achieve this operation atomically, the thread does a CAS with the tail of the queue. If the thread succeeds in doing so, it leaves the enqueue method. If the thread loses in the CAS attempt, it finds the new tail and repeats the process until it wins the CAS.

The enqueue implementation requires CAS operation to be done on next attribute of the newly created pointer_t object, hence the next attribute is made atomic to allow for this. If the ptr attribute of next in tail value read is not NULL, it means the tail is not pointing to the last node. In such a situation we need to atomically swing the tail pointer to point to the last node in the list. So the tail attribute of the queue_t class is made atomic to allow for atomic CAS operation. The ABA problem arised by using CAS operation is addressed by the use of count attribute of the node.

When a thread comes to dequeue, it reads the head and tail. If both are equal either the queue is empty or the tail is lagging. It checks for emptiness in which it returns **INT_MIN** indicating that the queue is empty. If the queue is not empty, the thread helps the tail pointer to advance itself and retries to dequeue. To help the tail pointer, we use a CAS operation. If head and tail are different, then we note the value and try to swing the head to next node. If we succeed we exit the while loop and reclaim the space of deleted node. In case we fail, we come again to try to dequeue.

Optimistic Queue:

This is an improvement over MSQueue.The Optimistic Queue algorithm addresses an inefficiency in the MSQueue where it requires two successful CAS(Compare and Swap) operations in order to complete the enqueue method. This algorithm finishes both enqueue and dequeue method by using a single CAS operation.

This is achieved by logically reversing the direction of enqueues and dequeues. This idea can cause problems because dequeues cannot traverse the list backwards to remove the element. So the algorithm uses a doubly linked list instead of a singly linked

list used in MSQueue algorithm. There is a fixList method which is called when we find inconsistent prev pointers in the dequeue method.

The following classes are used for implementing Optimistic queue:

```
class pointer_t
{
    public:
    node_t *ptr;
    int tag;

}
```

```
class node_t
{
    public:
    int value;
    pointer_t next;
    pointer_t prev;
}
```

```
class queue_t
{
    public:
    atomic<pointer_t> tail;
    atomic<pointer_t> head;

}
```

The queue is a **doubly linked list** of **pointer_t** objects.

When the enqueue() function is executed by a thread, it first creates a new node and sets the required value in the value attribute. Then it enters a while(true) loop where it first reads the tail and copies it to check_tail. Then it sets node's next to check_tail with tag incremented by one. Then it tries to CAS the tail to point to this new node. If the CAS succeeded, then we can say that the node is inserted into the queue. If not, the thread retries to enqueue because it is in a while(1) loop.

When the dequeue() function is executed by a thread, the thread enters a while loop for retrying in case of a failure. First it reads the current head and tail and copies them to check_head and check_tail. Then it stores the first node's prev in the variable "firstnodePrev". Now if the head was not changed and the queue is not empty, the method checks the tags of firstnodePrev and head. If they are not equal, the fixList method is called and the thread retries the dequeue. Otherwise the thread reads the value of 'firstnodePrev' and stores it in val. Now it tries to CAS the head to point to the 'firstnodePrev' with incremented tag. If the CAS succeeds, then the node is freed and val is returned.It the CAS fails, it retries the dequeue. If the queue is empty then it returns **INT_MIN.**

The fixList function takes the pointer to the queue, tail and head as the arguments. First it copies the tail to curNode. Then it runs a while loop with the condition (head==q->head) && (curNode != tail). While this condition is true, it reads curnode's next and fixes it's prev. Then it updates the curNode. Ultimately fixList helps the dequeue function.

Basket Queue:

This queue is an improvement over MS queue and offers more parallelism when there are multiple enqueue operations by creating a basket to comply with pending enqueue calls. It adds the elements in a random order to this basket contrary to expectation. Also multiple baskets could exist at the same time and operations in these baskets could be executed in parallel, offering more parallelism as mentioned earlier.

Compare-and-swap(CAS) is used to put overlapping enqueue elements in the same basket. CAS is applied to the queue's tail pointer and among all the overlapping threads, only one of them succeeds and remaining threads which have failed will be sent into the basket created by the winner thread.

The following classes are used for implementing basket queue:

```
class pointer_t
{
public:
node_t* ptr;
bool deleted;
int tag;
}
```

```
class node_t
{
public:
int value;
atomic<pointer_t> next;
}
```

```
class queue_t
{
public:
atomic<pointer_t> tail;
atomic<pointer_t> head;
}
```

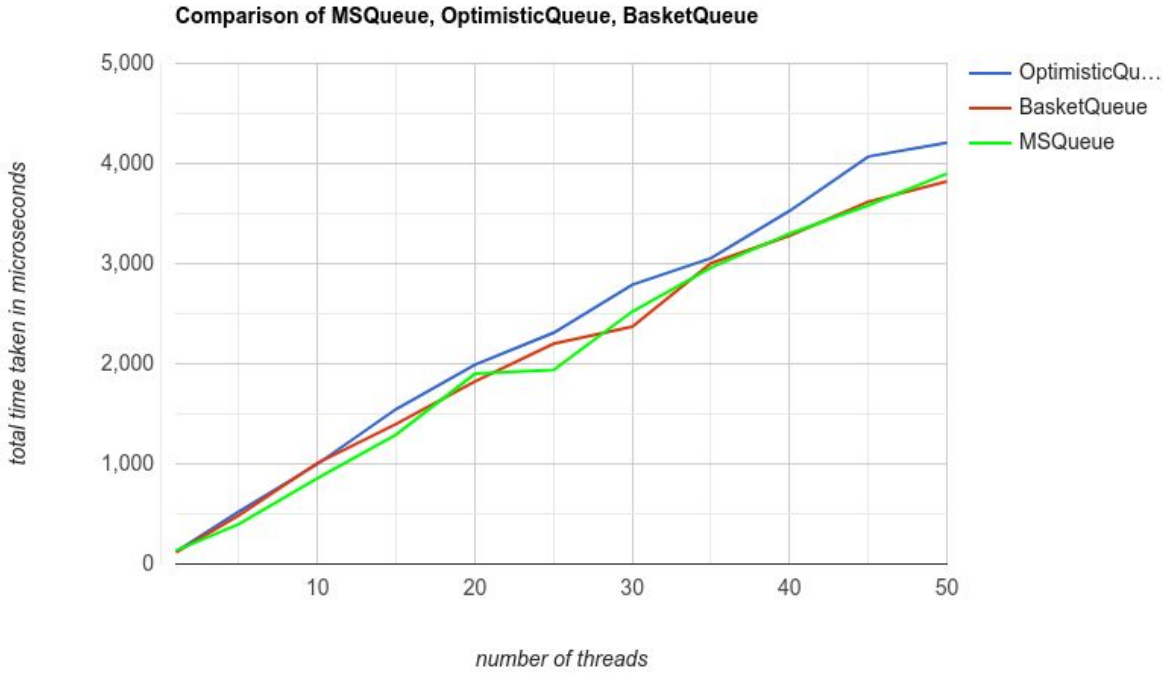
Whenever a thread invokes the enqueue method, the tail is read and checked if it is the last node. If it is the case, a new node will be tried to link to the last node. If there are no threads trying to enqueue or this thread is the winner of CAS, the thread makes sure that the tail of the queue is pointed to the node inserted by it. However if this thread is not the winner, it will insert the node(it wanted to enqueue) in the basket. The tail of the queue is also updated by searching the last node. Also we make sure that a new node can be inserted next to another unmarked node by using the deleted field in pointer_t class mentioned above.

Whenever a thread invokes the dequeue method, the current state of the queue is examined. Whenever the queue's head and tail points to the same node, there could be two possibilities.

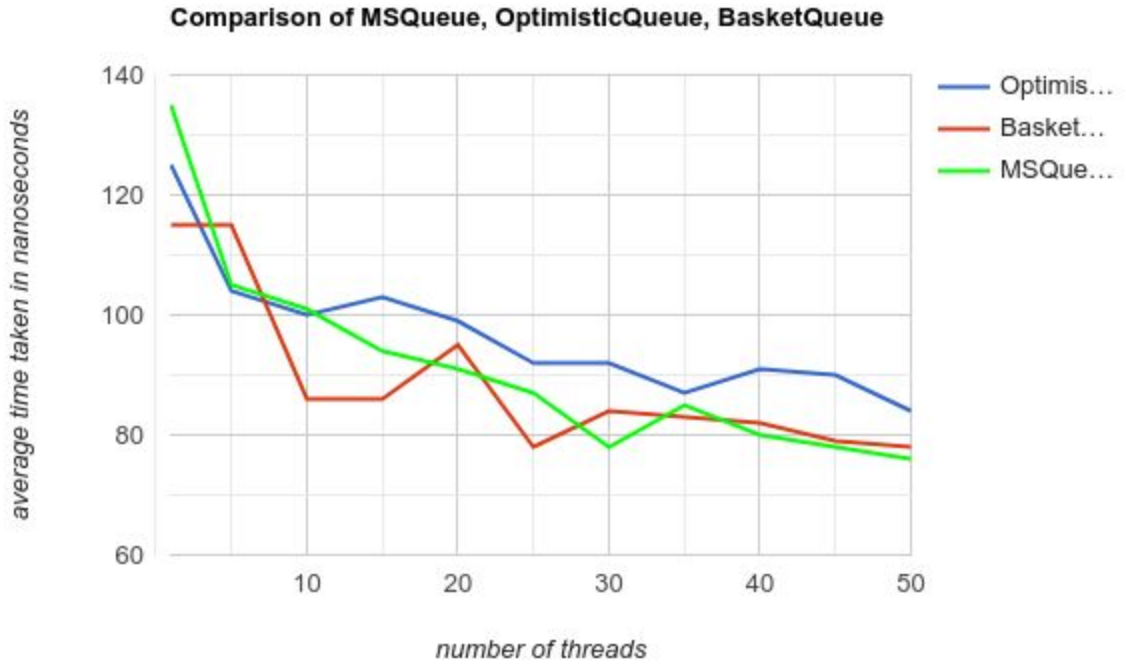
a. The tail is not updated : In this case, it searches for the last node, tail will be updated.
b. Queue is empty and there is nothing to dequeue. Then the thread returns INT_MIN.

But if the queue's head and tail don't point to the same node, then an unmarked node is searched. If it is found, then it's value is read and stored (to return) and then the thread tries to delete it logically.(by marking the bool field). Then the thread calls a method called free_chain which tries to update the head of the queue and delete(we make memory used by those completely by delete in C++) all nodes between old position of the head and the new one.

Comparison of performances of the queues:



Graph 1



Graph 2

- Graph 1 shows how total taken for n*1000 operations(each operation could be either an enqueue or a dequeue) varies with n.(where n is the number of threads). Basically each thread performs thousand operations(either an enqueue or a dequeue) and total time for all the threads to complete their respective thousand operations is captured and plotted against the number of threads.

- Graph 2 shows the average time taken for each operation.(each operation could be either an enqueue or a dequeue) varies with n. (where n is the number of threads).

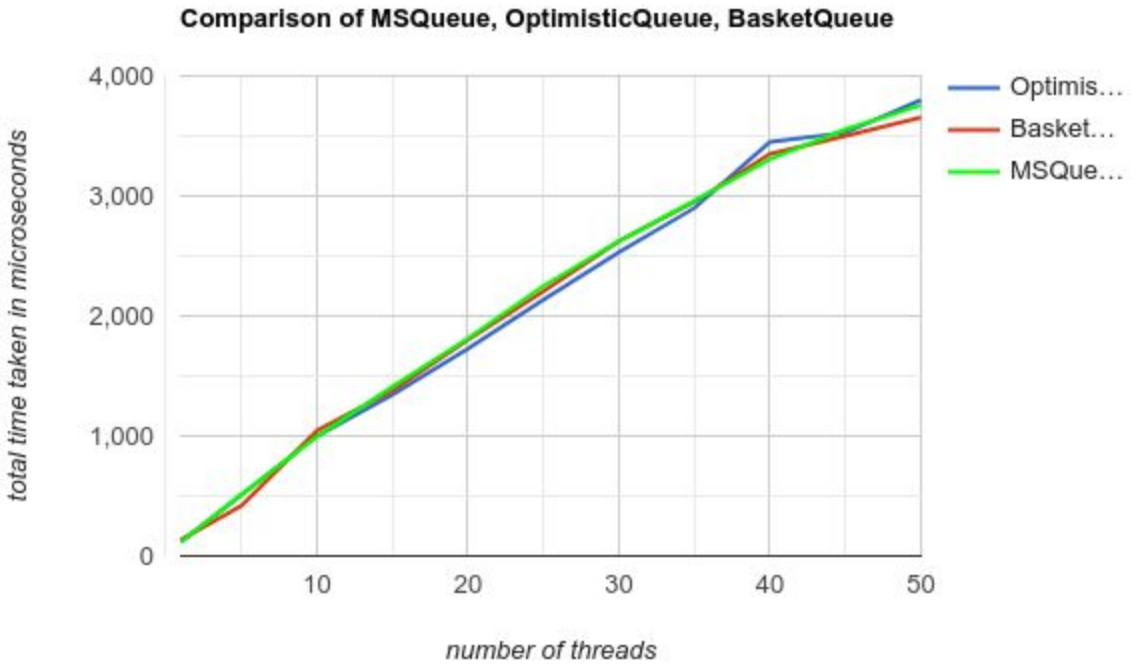
- As we can see from the graph 1, initially the total time taken for n threads increases fast but as the number of threads gets increased, the increase in total time taken is not much. The same thing is reflected in the graph 2. As the over time for execution of all operations is not increasing by much as the number of threads increases, the average time taken for each instruction to get executed shows a decreasing fashion which is reminiscent from graph 2.

- In graph 1, initially the basket queue is a bit slower than the optimistic queue but as the number of threads increases, the basket queue is faster. Because parallelism is attained in a basket queue using baskets upon high contention

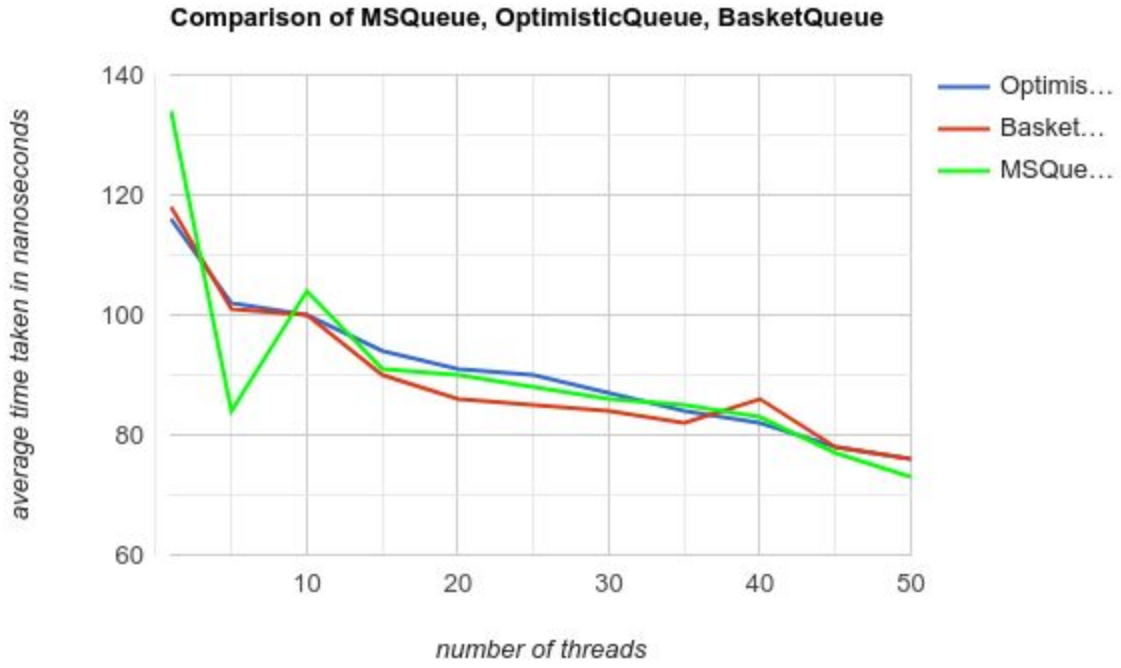
which is usually the case with high number of threads. Optimistic algorithm have only one CAS operation in dequeue when compared to two in MSQueue, this gives some efficiency to the Optimistic queue which might be reason why it is faster than MSqueue initially.

Among the thousand operations performed by each thread, in every iteration it randomly chooses one method among enqueue or dequeue. We've also computed other graphs where each thread, chooses enqueue and dequeue alternately.

These are the obtained graphs



Graph 3



Graph 4

- Although in graphs 1,2, each thread randomly chooses one method among enqueue or dequeue in each iteration and in graph 3,4 each thread chooses enqueue and dequeue alternately, both graphs are very similar. The same was expected assuming randomly selecting enqueue or dequeue would be of equal probability. The graphs explain the same.