CS5300: PARALLEL AND CONCURRENT PROGRAMMING, Fall 2020

Project: Lock-Free Queue

SAI HARSHA KOTTAPALLI

CS17BTECH11036

GOAL

We try to implement Lock-Free Queue from the paper "Lock-free Fill-in Queue" and analyze its performance by comparing it with a coarse-grained Queue with locks. We also propose an improvement to the algorithm in the paper for a bounded Queue.

INTRODUCTION

Queues are one of the fundamental data structures used in Parallel and Concurrent systems. One such application where it can be used is in handling multiple concurrent producers-consumers scenarios.

Though most of the lock-based queues are easy to implement and satisfy the correctness property, they also result in delays, deadlock, contention, etc. Hence, lock-free algorithms started to become popular as they would also provide a boost to the performance. More importantly, the algorithm doesn't stay in a deadlock wherein the previous scenario if the thread which acquired lock has crashed, then the whole queue is in deadlock. The challenge faced in developing lock-free-based implementation is to ensure the correct property is maintained, i.e., the FIFO property of the queue.

For this purpose, we implement lock-free queue using an atomic operation compare and swap (CAS). The algorithm also eases the enqueue and dequeue operations by reducing the use of CAS operations.

The paper provides us with a method to implement a bounded queue using a linked list and extending it to an unbounded queue.

BOUNDED QUEUE - PAPER

The paper proposes an algorithm for a bounded queue with the help of CAS operations to support multithreaded execution. It uses a circular linked list of k nodes(size of the queue) for its implementation with the objective that it is able to get the benefits of a array-based implementation that they dont have to keep creating new nodes and reuse when necessary.

Algorithm:

Initialization

```
// QNode
class Node {
public:
  atomic<bool> empty; // Flag
  int data;
  atomic<Node*> next;
};

// create k dummy nodes
for(int i = 0; i < k; i++) {
  Node* newNode = new Node();
  newNode->data = 0;
  newNode->empty = true;
  newNode->next = NULL;
  if (i == 0) {
      head = newNode;
  } else {
      tail->next = newNode; // previous node will point to this new
                            // node.
  }
  tail = newNode;
}
```

```
tail->next = head; // now cicular queue is complete
tail = head; // for init, both tail and head should point to same
             // node
```

Here, data represents value of the node.

Empty flag is to check where the node is currently empty or has data.

Finally, next point to the next node in the circular queue.

The initialization is pretty straightforward, we end up with a circular linked list of size k.

The queue uses two points head and tail to keep track of elements in queue.

Enqueue:

```
bool enqueue(int data, int th_id) {
  bool localFirstTime = true;
  Node* localTail = tail; //localTail is local
  while(true) {
      // Try any dummy node
      if(localTail->empty.CAS(true, false)) {
      // if tail points to empty dummy node
           localTail->data = data;

     // if previous enqueue is still ongoing, wait for it to finish
           while(tail != localTail) ;
           tail = localTail->next;

           return true; // as enqueue has ended successfully
      } else { // a concurrent enqueue has successfully enqueued, so
              // this has to try the next dummy node
           localTail = localTail->next;
           if(localTail == head)
                return false; // queue is full
      }
}
```

The tail pointer keeps getting updates as new items are being added into the queue until its full upon which it returns false, indicating that is was full and hence unable to perform enqueue operation.

First we copy the tail value into a local variable localTail.
From localTail, we start finding a node with was empty so that we can fill it until we come back to head again. Now everytime we check if node is empty we do so using a CAS operation so that if n threads compete here, only one thread wins, i.e., completes the CAS operation to return true and successfully updates the node with the value. It then updates the global tail pointer with the localTail's next node because this was the latest enqueue operation which was successful so that next node would be potentially free and finally returns true.

Now as for the node which couldnt acquire the node(CAS operation had returned false), it will proceed to the next node by updating its localTail and then cchekc again if the next node is free through CAS, and so on.

It maybe possible that the threads(say A) had won the CAS race and got access to a node, and another thread (say B) also competed but CAS returned false. Now lets say A became slow for some reason while thread B quickly executed CAS on the next node and updates the tail pointer. Now A also updates tail pointer, but its not the the correct node since B has already filled it.

Hence, we add the while loop to check if tail is indeed pointing to localTail or not, busy waiting till then. If it was same that means all previous enqueues have successfully completed and this thread can update the tail pointer.

Though the author hasnt mentioned why this is required, at first look this doesnt seem necessary as even if tail didnt point to latest node, it can still reacch it because the next nodes would be full when in an old thread updated tail very late. But this not true, this is necessary because if not it would break the correctness.

Now let us assume A, B, C had enqueued in the 1, 2, 3 and size of queue is 5.

Queue:

1 -> 2 -> 3 -> X -> X

here, Lets say C has updated tail while A is very slow and hasnt updated tail yet.

Now, there were two dequeues, so state of queue becomes:

X -> X -> 3 -> X -> X

Now A updates the tail pointer, and sets the first node as tail. Now enqueue(4) would result in state:

4 -> X -> 3 -> X -> X

which is clearly violating queue property. Hence, the while loop is essential.

Now finally when localTail is head, that means the queue is full so we return false.

An important point to note here which is also not mentioned in the paper is that, when queue is full and a enqueue operation occurs, it would check head which if we assume was still non-empty if no dequeues happened, it doesnt return false, straight away, it would loop over the entire list again until it reaches head. Though this might not be suitable for very large queue, it is also beneficial as during the time it completes a circle, some dequeue operation might have given space for it and hence rerducing the total number of FullQueue messages/ return false by the operation.

Dequeue:

```
int dequeue(int th_id) {
  Node* localHead = head;
  while(true) {
     // Now if localHead points to non Emptynode
     if(localHead->empty.CAS(false, true)) {
           int data = localHead->data;
           while(localHead != head)
                ; // wait
           head = localHead->next;
           return data;
     } else {
     // If CAS fails because of a concurrent successful dequeue
     // try the next node
           localHead = localHead->next;
           if(localHead = tail->next) {
                return INT_MIN; // queue is empty
             }
     }
  }
  // the below line is not recheable
  return INT_MIN;
}
```

The dequeue operation works in a similar way as enqueue except that one successful dequeue we return data instead of true, and INT_MIN instead of false.

Here instead of finding nodes which are empty, we do CAS to find node which is non-empty, the first thread to execute it would update node to empty and thereby makes other thread move to next nodes. Here too we need a while loop so that previous dequeues are completed before the thread updates Head pointer.

Here too, when queue is already empty, the thread would check all nodes in a circle before returning INT_MIN if there were no enqueue operation in between.

BOUNDED QUEUE - PROPOSED

I had an idea to use an array of nodes instead of a linked list as array accesses are usually faster, to check if there is any improvement in the performance. Hence I updated the above algorithm which uses a circular array queue.

Initialization

```
class Node {
public:
  atomic<bool> empty; // Flag
  int data;
};

Queue(int k) {
  q = new Node[k];
  size = k;
  // init k nodes
  for(int i = 0; i < k; i++) {
     q[i].data = 0;
     q[i].empty = true;
  }
  head = 0;
  tail = 0;
}
```

Here, we create an array of k nodes and later apply mod k to all arithmetics done on tail and head.

```
bool enqueue(int data, int th_id) {
  int localTail = tail; //localTail is local
  while(true) {
      // Try any dummy node
      if(q[localTail].empty.CAS(true, false)) {
           // if tail points to empty dummy node
           q[localTail].data = data;
           while(tail != localTail) // if previous enqueue is still
                                    //ongoing, wait for it to finish
                ;
           tail = (localTail + 1) % size;
           return true; // as enqueue has ended successfully
      } else { // a concurrent enqueue has successfully enqueued, so
              // this has to try the next dummy node
           localTail = (localTail + 1) % size;
           if(localTail == head) {
                return false; // queue is full
           }
      }
  }
  // the below line is not reachable
  return false;
}
```

```
int dequeue(int th_id) {
  int localHead = head;
  while(true) {
      // Now if localHead points to non Emptynode
      if(q[localHead].empty.CAS(false, true)) {
           int data = q[localHead].data;
           while(localHead != head)
           ; // wait
           head = (localHead + 1) % size;
           return data;
     } else {
     // If CAS fails because of a concurrent successful dequeue
     // try the next node
           localHead = (localHead + 1) % size;
           if(localHead = (tail + 1) % size) {
                return INT_MIN; // queue is empty
           }
     }
  }
  // the below line is not reachable
  return INT_MIN;
}
```

The working remains very simiar to the original algorithm but the only difference is that instead of tail and head pointers we use tail and head array indices. Since the next node is the adjacent, we simply use +1 and mod size to handle that.

We analyze later to check if there was any boost to the performance offered by this new algorithm.

UNBOUNDED QUEUE

The dequeue method remains the same as the one from paper, the enqueue method was updated.

```
bool enqueue(int data, int th_id) {
  bool localFirstTime = true;
  Node* localTail = tail; //localTail is local
  while(true) {
     // Try any dummy node
     if(localTail->empty.CAS(true, false)) {
           // if tail points to empty dummy node
           localTail->data = data;

           while(tail != localTail) {// if previous enqueue is still
                 ;              // ongoing, wait for it to finish
     }
     tail = localTail->next;
     return true; // as enqueue has ended successfully
     } else { // a concurrent enqueue has successfully enqueued
             // so this has to try the next dummy node
     Node* prevLocalTail = localTail;
     localTail = localTail->next;
     Node* newNode;
     while(localTail == head) {
          // No more empty dummy nodes and queue is full,then
          // create a new node
          if(localFirstTime == true) {
               localFirstTime = false;
               newNode = new Node();
               newNode->data = data;
               newNode->empty = true;
               newNode->next = NULL;
          }
          newNode->next = head;
          if(prevLocalTail->next.CAS(newNode->next, newNode)) {
               //Successful CAS and node is connected
               while(tail != prevLocalTail)
                    ;
               tail = newNode;
               return true; // as enqueue has ended successfully
          } else {
               prevLocalTail = prevLocalTail->next;
          }
     }
     }
  }
  // the below line is not reachable
  return false;
}
```

Now the extension handles the case where there are new enqueues when queue is full, but it is not lock-free anymore and is the reason why i was not able to run simulations on this to compare with others.

For example, lets say the queue is currently as shown below:

1 -> 2 -> 3 -> 4 -> 5, where head points to 1, tail points to 1, queue being full.

Now if there is only **1** thread performing enqueue and no other thread is doing any dequeues, following the algorithm similar to bounded case it would go through all the nodes and finally reach the head upon which it enters the while loop.

Now it will create a new node with prevLocalTail pointing to node with data 5. The CAS is also executed without any issue since there is only one thread and it links the new node between prevLocalTail and head. But now, it is stuck in next while loop, i.e.,

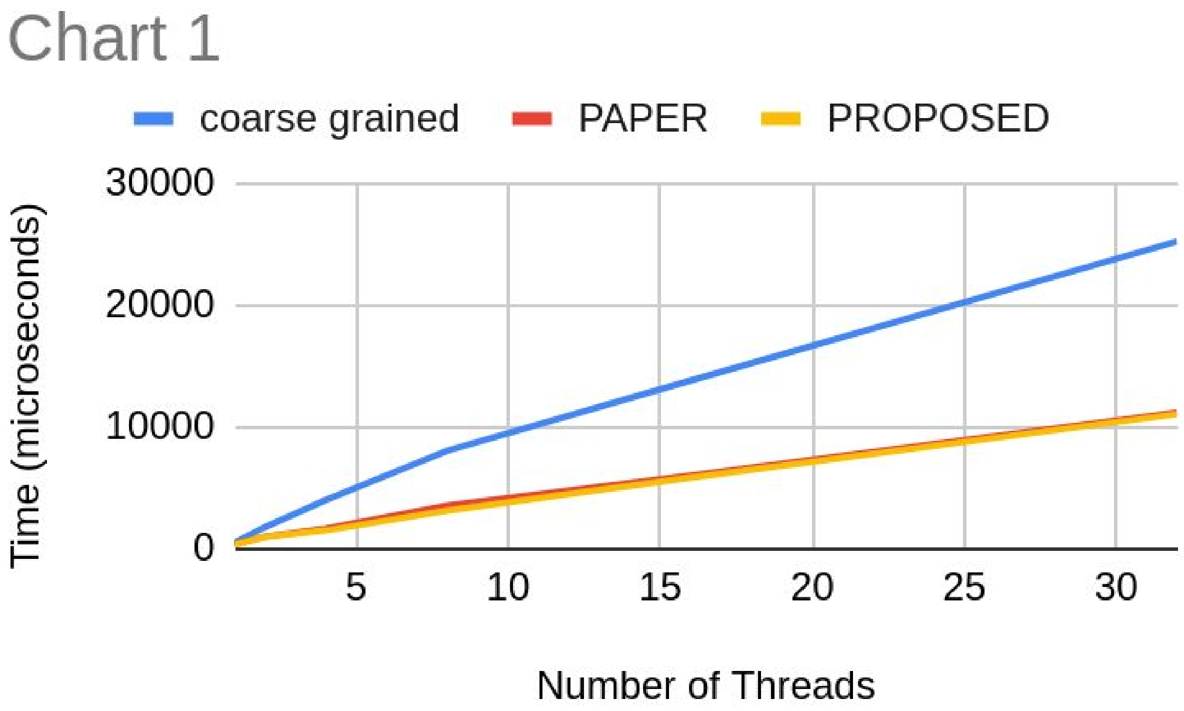
```
while(tail != prevLocalTail) ;
```

now prevLocalTail is pointing to 5 while tail points to 1, this will not be updated unless and until some other threads do more enqueue orr dequeue operations and hence, this thread keeps spinning forever. Therefore, this algorithm was not suitable for analysing through a simulator since it could go on forever. This is thereby, not lock-free, since it doesnt even guarantee obstruction free.

ANALYSIS

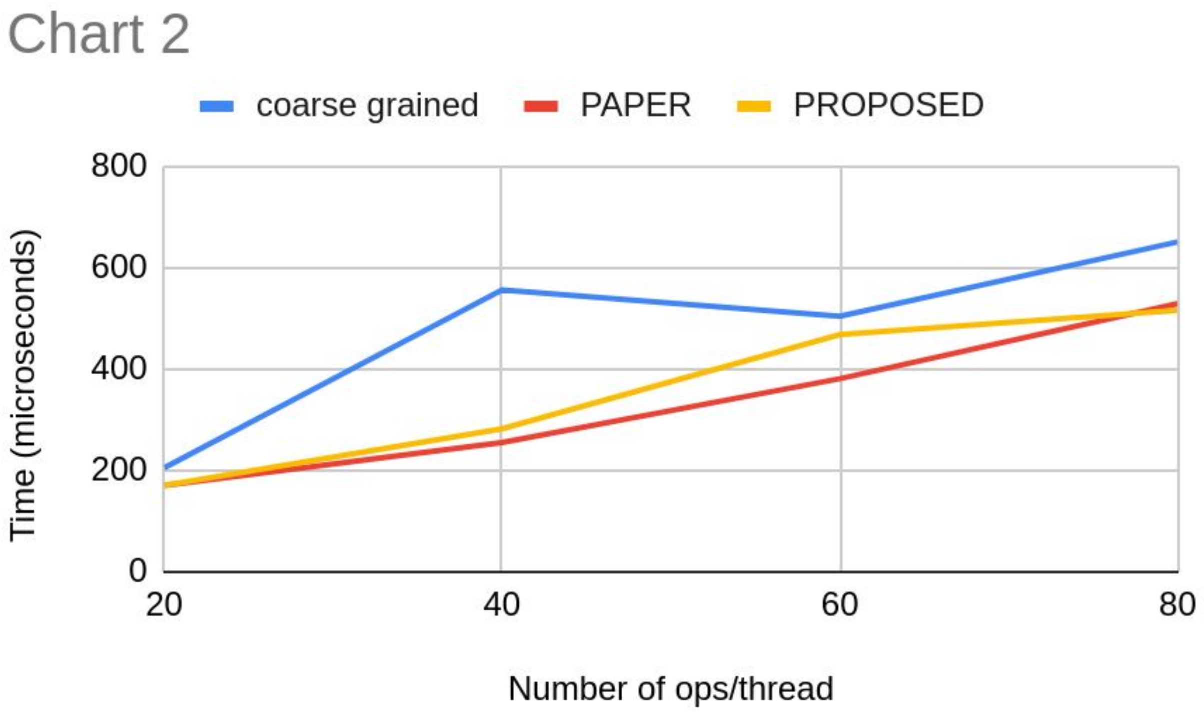
For comparison, i will compare the bounded queue from paper, proposed and another one implemented using a coarse grained lock. Since, enqueue and dequeue operations are quite fast, we measure time in milli seconds
For the simulation. i set a seed so that all three algorithms get the same random numbers, and accordingly will enqueue or dequeue with 50% probability.
I also tried to check how many times it failed to enqueue due to queue being filled, it seems there is no difference between the algorithms in this scenario. Maybe on a very high number of threads we may observe differences but it is out of the computational power of my laptop.

1. For a fixed number of operations from a thread while increasing the number of threads.
   queue size = 30,number of ops/thread = 50.

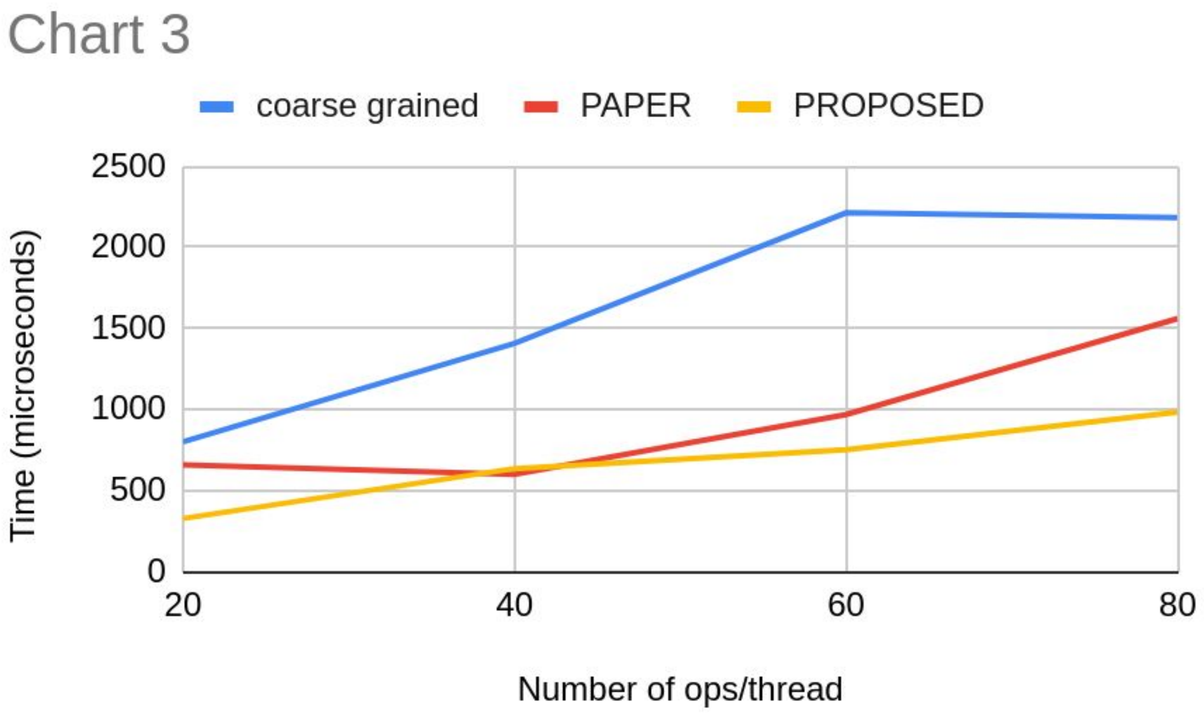


As expected, it performs better than coarse grained since the algorithm is lock free. It seems the modifications proposed make it a little bit quicker but it's only negligible. Maybe on a very high number of threads we may observe differences but it is out of the computational power of my laptop.

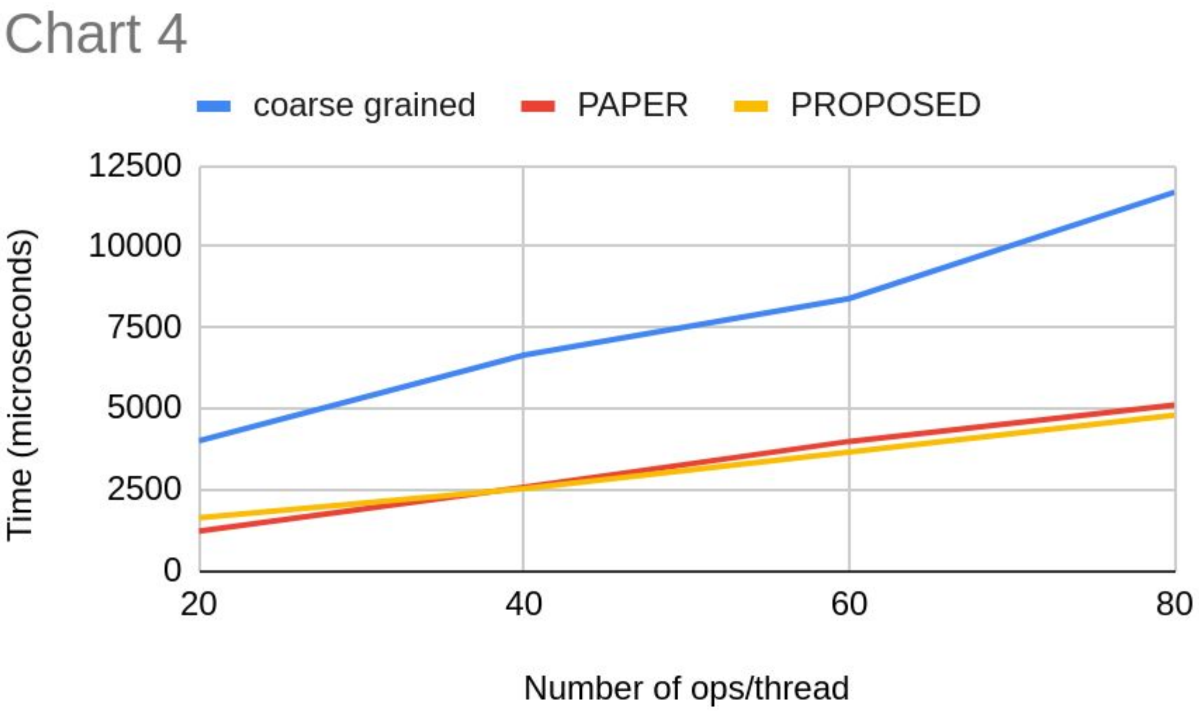
2. For fixed number of threads, changing number of ops/thread
   queue size = 30, number of threads = 1



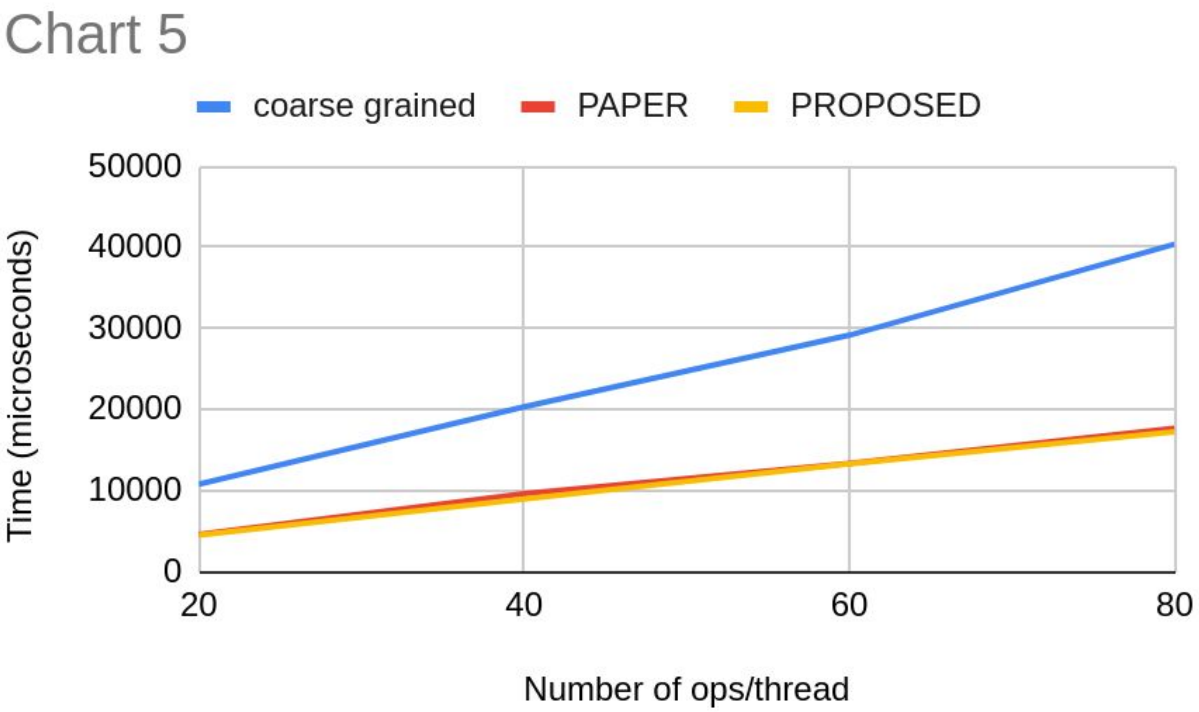
queue size = 30, number of threads = 2



queue size = 30, number of threads = 8



queue size = 30, number of threads = 32



Similar to the 1st graph inference, the modifications proposed make it a little bit quicker but it's only negligible.