Programming Assignment 1: Implementing BOCC and FOCC algorithms

Goal: The goal of this assignment is to implement the BOCC and FOCC algorithms studied in class in C++.

Implementation Details:

The implementation of both BOCC and FOCC are similar with the main difference lying in the tryCommit function.

File input variables:
No. of threads(nThreads) = 10
No. of variables(m) = 10
No. of total transactions(nTx) = 500, 600, 700, 800, 900, 1000
Lambda = 20

Data Structures:
map<int,int> status -> for maintaing status of the transactions which enter the execution cycle.
map<int,vector<int> > rS -> for maintaing readset of each transaction
map<int,vector<int> > wS -> for maintaing readset of each transaction
map<int,vector<int> > rL -> for maintaing read list of all variables acting as buffer

Mutex Locks:
vLock, pLock, mLock, cLock, idLock

For each operation:
begin_trans() = acquire idLock -> assign tx_id -> release idLock -> acquire vLock -> initialize rS, wS, status -> release vLock
read(int tx_id,int dx) = acquire vLock -> check tx status -> add to rS, rL -> release vLock
write(int tx_id,int dx) = acquire vLock -> check tx status -> add to wS -> release vLock
cleanup(int tx_id) = acquire cLock -> remove tx from rS, wS, rL, status -> release cLock
tryCommit(int tx_id) = checks the current transaction with previous ones on the condition specified with BOCC, FOCC-CTA, FOCC-OTA

BOCC = Checking intersection of write-set with read-set of transactions by comparing with read-list of variables in wS if empty, otherwise abort

```
if(rL[wS[tx_id][i]].size() > 0)
        {
           cleanup(tx_id);//If intersection != null => Contradiction, then remove the transaction
           pthread_mutex_unlock(&vLock);
           return 0;
        }
```

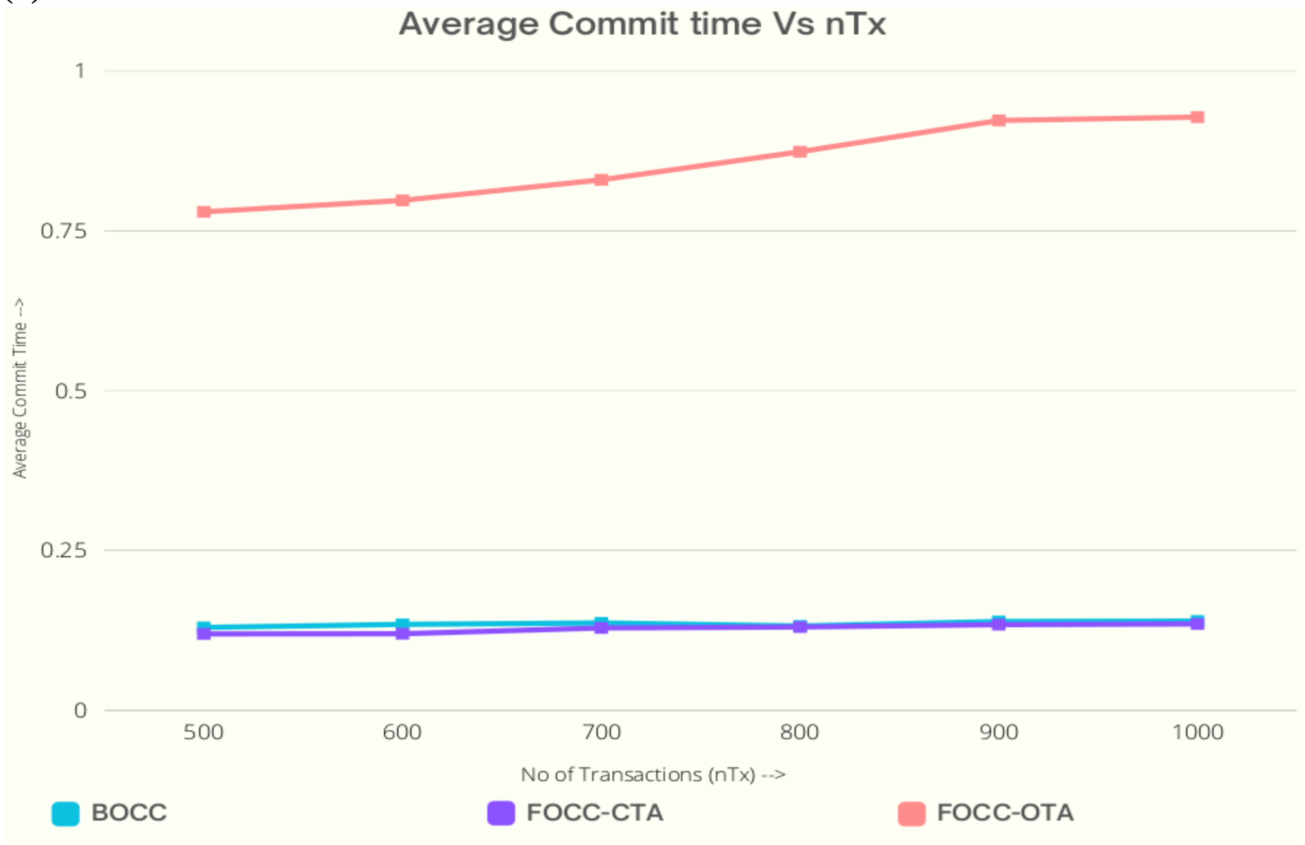
FOCC-CTA = Considering concurrent live transactions, comparing if read-list of variables in wS if empty, otherwise abort

```
while(status[i] == 0)//Considering concurrent live transactions
        {
           if(rL[wS[tx_id][i]].size()>0)
           {
            cleanup(tx_id);//If WS[j] intersection RS^n[i] != null => Contradiction, then remove
the transaction
            pthread_mutex_unlock(&vLock);
            return 0;
         }
      }
FOCC-OTA = Considering concurrent live transactions, comparing if read-list of variables in
wS if empty, otherwise abort the other transaction
int chkterm = rL[wS[tx_id][i]].size();
      if(chkterm > 0)//If WS[j] intersection RS^n[i] != null => Contradiction, then remove the
other transaction
      {
         for(j=0;j<chkterm;j++)
         {
            status[rL[wS[tx_id][i]][j]] = 1; //Setting status for abort
         }
      }
```

Average commit time = commitTime/nThreads
Average time taken by tx to commit successfully starting from begin_trans().
Average abort count = abortCountGlobal/nThreads
Average no. of tx aborted per commited tx.

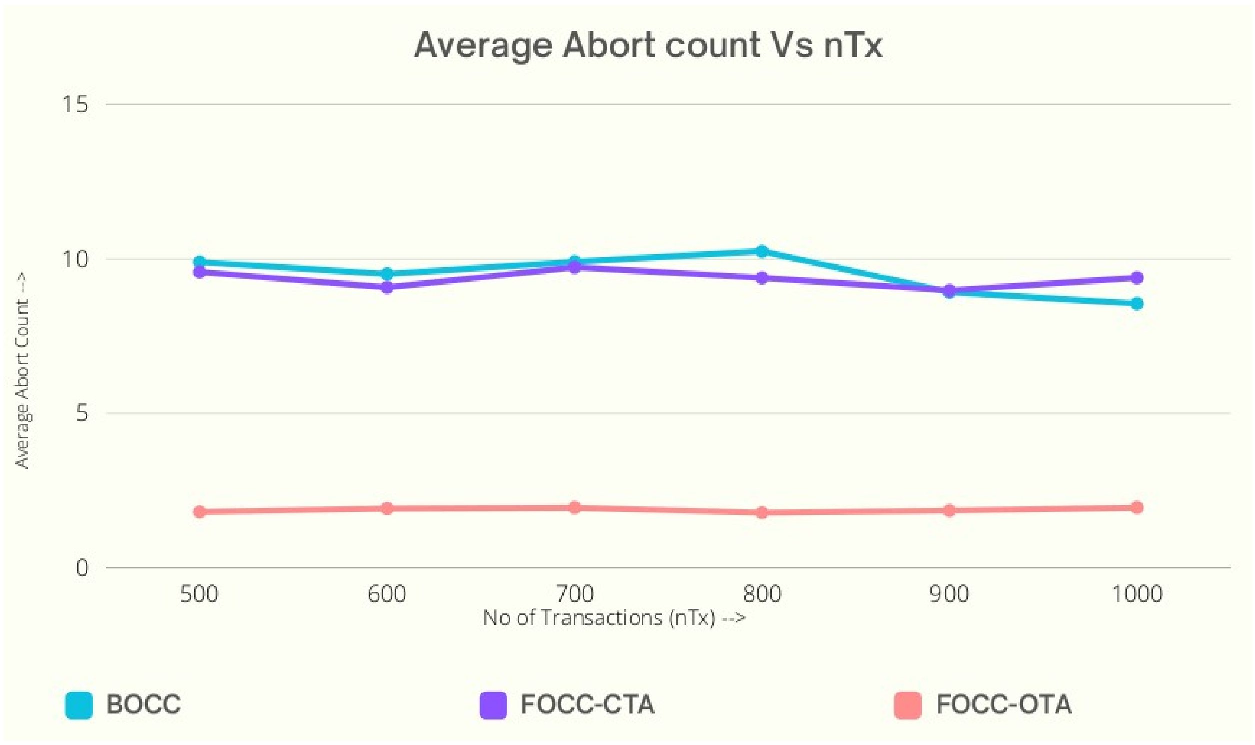
Graphs:

(1)



This graph plots Average delay to commit vs Total no. of transactions per thread.

(2)



This graph plots Average abort count vs Total no. of transactions per thread.

Observations:
Slight increase in average commit delay is observed with increase in no. of transactions.
This could be due to increase in time taken for book-keeping operations due to larger no. of transactions.
More increase in average commit delay can be seen with increase in no. of transactions for FOCC-OTA than compared to FOCC-CTA and BOCC. FOCC-OTA has greater slope in average delay graph. Average commit time for OTA is higher as it has to check for all intersection of wS with mutiple other transaction and abort the other transaction if condition gives true. But in CTA we find atleast one intersection and abort the current transaction.

Average no. of aborts remains roughly constant as we can see increasing and decreasing trends equally. This may be because the no. of aborts are not changing as rapidly as no.of transactions that has been increased.
FOCC-OTA shows better performance as it gives lower abort count (<5) than compared to FOCC-CTA abd BOCC (roughly aound 10).