Comparison of BOCC-CTA, FOCC-CTA, and FOCC-OTA Protocols

Tushita Sharva

March 29, 2025

1 Problem Statement

This project implements and compares the scheduling algorithms: BOCC-CTA, FOCC-CTA, and FOCC-OTA. The performance is measured in terms of:

- The average number of times a transaction is aborted before committing (abort count).
- The average time taken for a transaction to commit (commit delay).

Parameters

- n: Number of threads.
- m: Number of shared data items.
- $totalTrans$: Total number of committed transactions required.
- $constVal$: Parameter for simulating increment operations on data items.
- λ: Parameter for simulating complex operations in transactions.
- $numIters$: Number of read and write actions per transaction.

2 Program Design

The project consists of several classes implemented in C++.

2.1 Classes & Their Methods

2.1.1 Logger Class

Handles logging for output and debugging. Provides time-stamped log entries for transaction execution and validation.

2.1.2 Transaction Class

Maintains attributes of a transaction. Key members:

- **id**: Unique transaction identifier.
- **status**: Transaction state (active, committed, aborted).
- **readSet**: Set of data items read.
- **writeSet**: Set of data items modified.
- **localWrites**: Stores local modifications before committing.
- **start_time, end_time**: Timestamps for validation.

2.1.3 DataItem Class

Encapsulates shared data items and tracks transactions accessing them. I implemented separate classes for protocols that abort current transaction and one that aborts other transaction. This segregation was required because the current transaction needs the access to another transaction's status variable to be able to abort it instead of aborting itself. The only difference was wether the writeset stores only the transaction ids or the entire transaction struct. Key attributes:

- **value**: Current value of the data item.
- **readList**: Transactions that have read this item.
- **writeList**: Transactions that have written to this item.
- **highest_endTime**: This attribute is used only for implementation of BOCC protocol. **Whenever a transaction commits, it informs all data items in it's write set that it is the last transaction that committed and wrote to this data item**. We will see that by having this one extra attribute, we can avoid iterating end times of all transactions in validation phase of *BOCC*.

3 Validation logics for the protocols

BOCC:

- Whenever a transaction performs a read or *local* write on a data item, we insert the data item into the transaction's readset or write set.
- In try commit, we should check if our transaction's readSet's dataItems' writeList is empty or not.
- So we first acquire locks on each of the data item in our readset (union write set, but in given program flow readset = writeset).
- Once we acquire all locks on data items, we need to check that the readlist of the data item is
  - either empty
  - or all the transactions in it's set occurred before this transaction started.
- In simple words, if there is any one transaction that is concurrent with current transaction, and has already written something which I read, I should abort myself.
- We check if the highest end time is less than the transaction's start time. If not, it means there is at least one transaction that had concurrent read phase with me. In that case, atleast the writeList of that data item should be empty.
- If not both, we release all locks and we abort.
- If not aborted, we make the writes global and commit.

FOCC-CTA:

- Whenever a transaction performs a read or *local* write on a data item, we insert the data item into the transaction's readset or write set.
- We also insert the transaction into the data item's readlist when a transaction performs a read.
- For validation we should check if our writeset's dataitems' readlist is empty or not.
- So we first acquire locks on each of the data item in our writeset (union read set, but in given program flow readset = writeset).
- Now that we acquire the locks, by the time we release locks and allow other transactions to access this data item, we will already be committed or aborted. We won't be concurrent anymore, so we can now remove ourselves from the readlists of all data items we read till now.

- Now we check if there are any conflicts, and if there are any, we abort ourselves.
- If we are not aborted, we make our writes global and then release all the locks on data items. Else, we release all the locks and simply return.

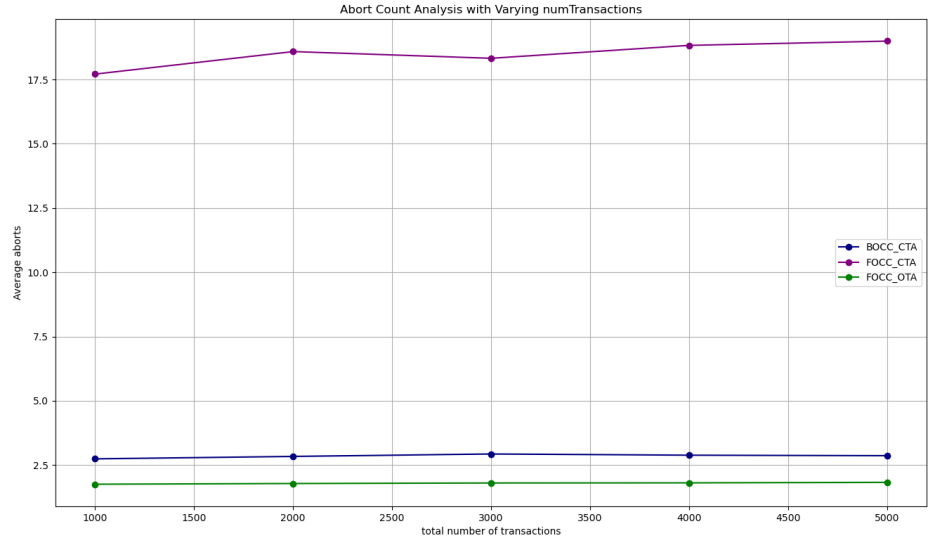
FOCC-OTA:

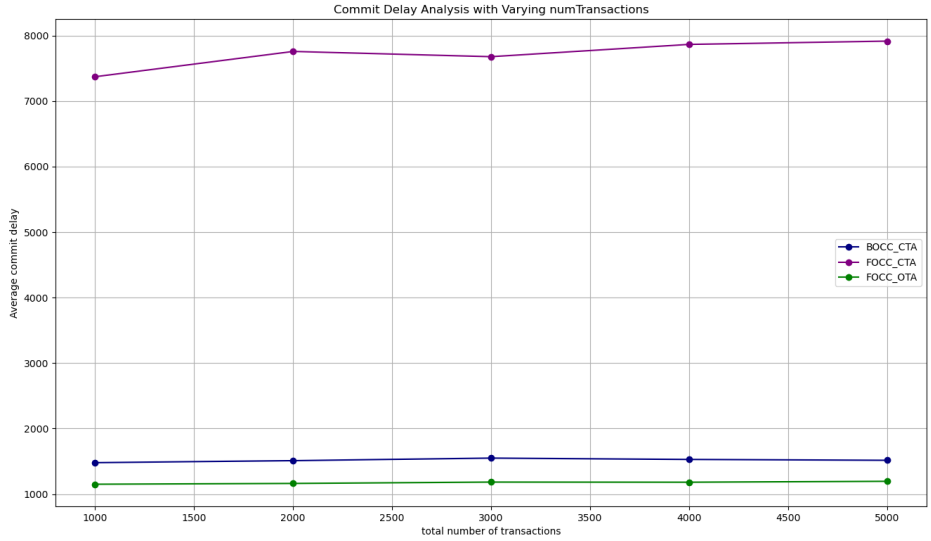
- Whenever a transaction performs a read or *local* write on a data item, we insert the data item into the transaction's readset or write set.
- We also insert the transaction into the data item's readlist when a transaction performs a read. Note that unlike FOCC-CTA, we will insert pointer to transaction and not just transaction id, because we will have to access a transaction's status variable to abort it.
- For validation we should check if our writeset's dataitems' readlist is empty or not.
- So we first acquire locks on each of the data item in our writeset (union read set, but in given program flow readset = writeset).
- Everytime we acquire a lock, if it waited, it means that it waited for some transaction to evaluate, and therefor, there is a chance this transaction's status is set to abort. So we check if there is an abort before acquiring lock on each data item.
- If after acquiring all locks, if this transaction did not abort, then it will continue and probably set others to abort, if at all there are conflicts.
- Now that we acquire the locks, by the time we release locks and allow other transactions to access this data item, we will already be committed or aborted. We won't be concurrent anymore, so we can now remove ourselves from the readlists of all data items we read till now.
- Now we check if there are any conflicts, and if there are any, we abort that transaction.
- If we are not aborted, we make our writes global and then release all the locks on data items. Else, we release all the locks and simply return.

4 Experimental Results

Results from executing `bocc`, `focc_cta` and `focc_ota` will be analyzed. We compare in terms of Average abort count per transaction, and Average commit delay.

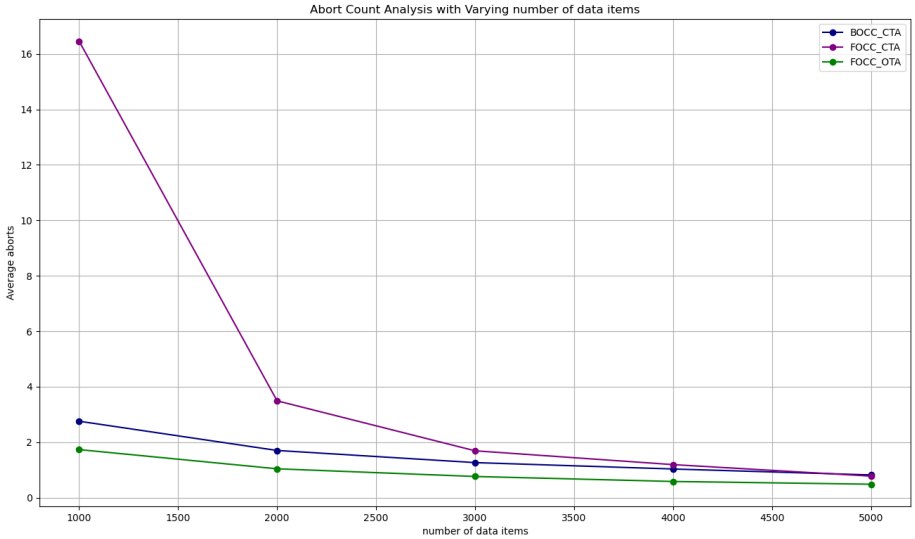
4.1 Increasing Transactions using 16 threads

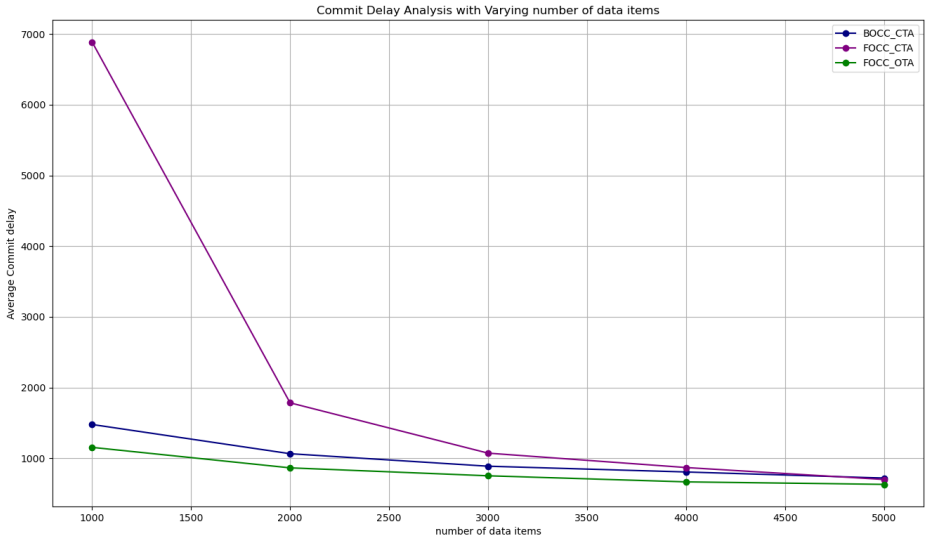




Explanation We can see that FOCC performs the worst, because of more number of unnecessary aborts. FOCC-OTA performs the best. Due to the default scale chosen by python due to the difference in results for best and worst performing schedules, we are unable to observe fluctuations in graph. However, we can say that by increasing transactions, the average number of aborts and commit delays increase, which is also intuitively correct.

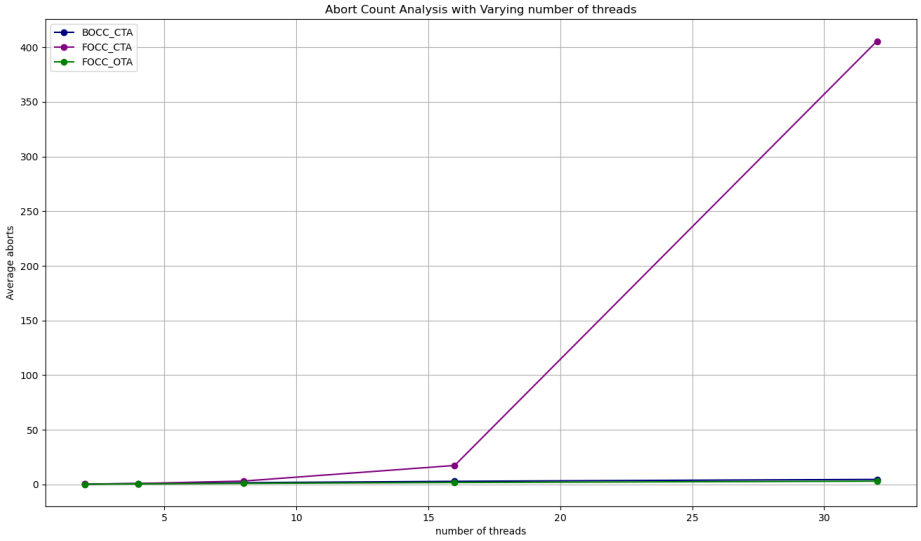
4.2 Increasing data items using 16 threads

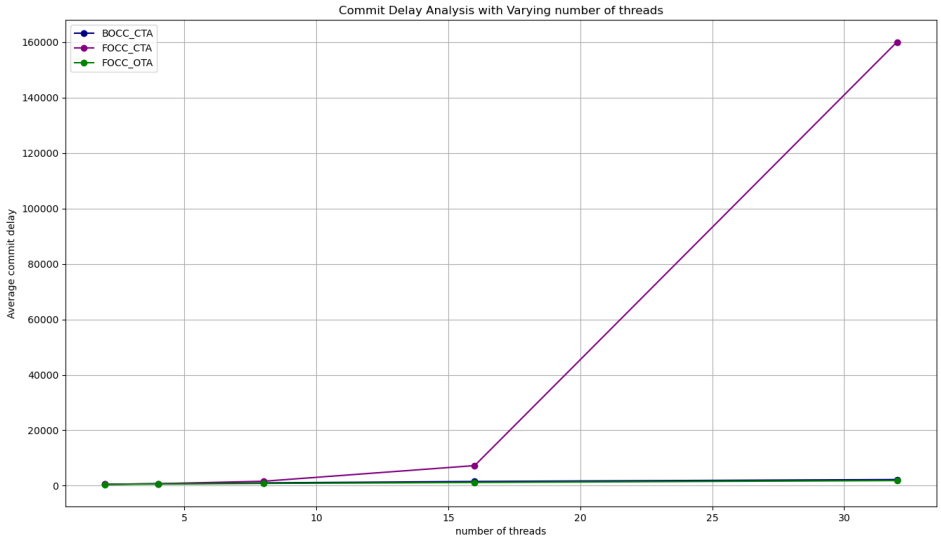




Explanation By increasing the number of data items, we can see that the abort count and commit delays are rapidly decreasing. It is because when there are more data items, and when we are sampling the data items, we are decreasing the likelihood of same data item being picked by concurrent transactions, which reduce conflicts and thus reduce the abort counts and commit delays. We also see that FOCC-OTA performs the best because it reduces un-necessary aborts and also saves time. A transaction aborts every conflicting transaction at a time instead of each of them realising themselves that they need to abort.

4.3 Increasing number of threads





Explanation Usually we see that increasing threads improves performance because of a greater degree of concurrency. But in this case, by increasing the threads, we are increasing the number of concurrent transactions, not dividing work but we are creating more competition. When we have two threads, affectively only 2 transactions are running paralelly - chances they conflict are low, number of items to check are low and so on. Hence the result. If we were to measure the overall time taken for the programs to execute instead of abort count or commit delays, the result would've been different. Also, we can see that FOCC-CTA doesnot scale well with the increasing number of threads, which is expected.

5 Conclusion

From our experiments, we observe that FOCC-OTA consistently outperforms BOCC-CTA and FOCC-CTA in terms of reducing abort counts and commit delays. The primary reason for this is FOCC-OTA's proactive approach in resolving conflicts by aborting conflicting transactions early, rather than relying on individual transactions to detect conflicts themselves.