Report

System Overview:

In this implementation. The client is a process which has 8 threads and run in close loop. Client use TCP socket to send request to the server.

The servers communicate each other with MPI message. And each server is both coordinator and participants and perform phase1a, phase1b, phase2a, phase2b.

Client Details:

Client will send its request and a bit called is\_end. If is\_end is set True then the tcp\_recieve thread on server for that client will end. Which promise that there will be no orphan thread. Also a integer number which indicate the order of the message send by client.

Server Details:

Every thread in the whole system will have a global id which is called as uuid made up by 2 part (world\_rank, socket\_id). The total number of uuid is 40(8 \* 5).

Receive request from client:

After receiving 8 sockets, there would be 8 thread continually receive request one after another from client in a close loop mode. After receive request it will call phase1a() to send message to the corresponding nodes, phase1a return immediately after send the message. In my second assignment I didn’t let the tcp\_receive() loop halt, so The TCP\_receive method will go to the next loop, which will call phase1a again. In this assignment I halt the tcp\_receive thread by checking if the comming request order is equal to the current request order.

Server communication:

There is a mpi\_receive thread, whenever it receives message send from other server node it will get one thread from thread\_pool and run corresponding function on such thread to deal with the received message. For example, when receive phase1a message, it will run phase1b on one thread to deal with such information.

2pc Strategy:

For GET request, 1 phase is enough and if the node is local, send to local, otherwise send to the replica.

For PUT request, if one of the replica doesn’t give promise from phase1b message, I will send twoAMessage to each node let them abort such request.

For multiput request, the request send to nodes as a whole and let the server node which will be lock. Same as PUT if we can’t get all promise for all keys, we will run phase2a to each node to abort all.

Log:

For coordinator, I write prepare to lock message in log which is in phase1a before send oneAMessage.

And write commit or abort to log before send twoAMessage in phase2a

For participants, I write agree to commit or abort to the log which is in phase2b before send twoBMessage.

And write decision to log for the request before send twoBMessage in phase2b

I think to make it easy to debug and recovery I’d like to give all message a uuid instead of thread for later time.

Tricks:

1:

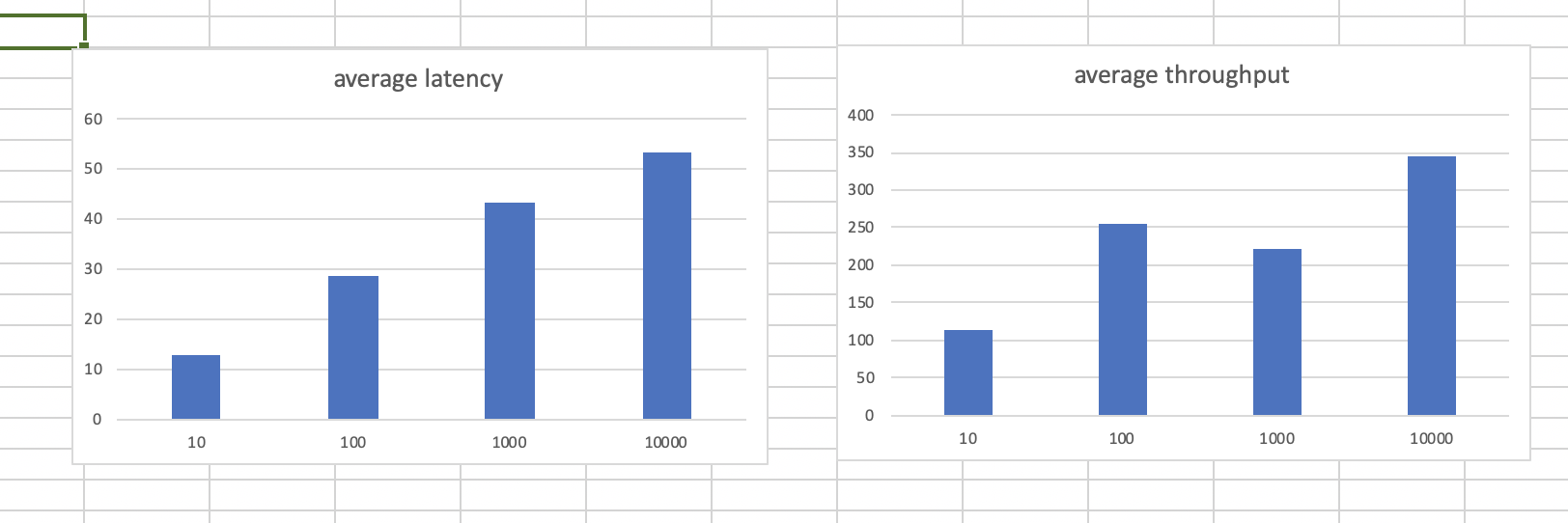
since every lock should be release by the same thread. So, when we do the lock operation on one server node for one key, we are not actually lock it but try\_lock the corresponding mutex first and then use a hash\_table to maintain and deal with “lock” and “unlock”. The key of this hash\_table is the uuid, because anytime there will be at most 1 request for each client’s one thread, because of the close loop.

The actual mutex will be release in phase1b anyway. So that our system will never have dead lock problem. And the fake lock will be release after receiving twoAMessage of that request.

2:

because a server may process multiple responds for the same uuid at same time. for example, a multi-put request will send to at most 5 nodes, these nodes will all responds oneBMessage, when the server receives all 5 message it will draw 5 thread to process these messages, so it need a local lock for each uuid.

Performance:



The latency is ms on y-axis,

The throughput is number of operation in 1s