Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 08
Lecture - 45

Lec 45:Proving Correctness (1)

Hello everyone. We are doing module 5 on scalable shared memory systems. This is lecture number 6 where we are going to discuss about proving correctness of the protocol. So quick recap about correctness. If you remember, we were discussing protocol has to work correctly and then it should be free from deadlock, livelock, and starvation. So these were our requirements.

And for ensuring the correctness of the protocol, we have to guarantee that it is coherent and consistent. So protocol should ensure that all the shared blocks across the system, they are either updated or they are invalidated before the modified value of a new value can be written into that data location. Right. So we have to ensure that all the blocks get invalidated or updated. Then we have the concept of serialization, that is when I am writing a data value to a block, all the nodes in the system should see thewrites to this location in the same order as this particular process.

So everybody should see the writes in the same order or the same serialized order, be it across different locations or for the same location. Sowrite serialization has to be maintained for coherence and consistency. And of course deadlock, livelock, and starvation freedom have to be achieved. So we can say that the first level that protocol ensures invalidation can be achieved if I am confident about the protocol and we are not going to discuss this aspect further. Serialization and deadlock, livelock freedom, become more complicated in my current setup because it is a large scale network.

We don't have a single entity called the bus, through which I could serialize transactions where I could have a small limit on the number of transactions which are happening at a given point of time. In the current scenario, we have a scalable network. So there are multiple transactions in flight at a time. And there are multiple copies and multiple transactions happening parallely and how do I still guarantee serialization? How do I guarantee correctness and deadlock freedom? So we need to handle this more carefully in a scalable network. So for correctness, we are assuming that the protocol ensures the relevant block and won't discuss this further.

We are going to tackle the serialization aspect in detail and definitely the other three properties. And we need to do this carefully because I have multiple copies of the block and there is no single agent. This is our main problem, that we don't have a single agent or one place which sees every relevant transactions. What was this agent in a simple system which we discussed? It was the bus because all transactions went onto the bus and bus was able to see what is happening. As I have several many processors, each sending a lot of requests and all of them probably could be directed to the single node.

For example, all of them request a similar block, they all go to the same home node and the home node might have buffering problems. It won't have space to house these many requests. Scalable systems, they will have high latency because you have to go through several hops and several intermediate nodes to implement the protocol. To solve this latency, we have looked at some protocol optimizations. And what are these optimizations handling? They are actually trying to reduce the number of transactions in the critical path.

How to do this? Either by having newer protocols or overlapping transactions. So I am going to parallelize or overlap certain actions of the transactions, so that my critical path becomes smaller. In doing this, we are having to do away with the strict request response nature which will eventually complicate my correctness requirements. So in a scalable system, high latency is there. To solve high latency, better protocols.

Better protocols make things in parallel. More parallelism means no strict request response, leading issues of correctness. So let us do the first one, serialization to a location for coherence. So a quick recap about what is serialization. If you recollect the bus-based system where all the requests were going onto the bus and we could string them together to get one serial order of actions happening on a particular data block. Right.

So we would say that processor 1 did the read, then 2 did the write, then 3 did the read and so on. So we could derive a serial order related to one particular location because I had a single entity called the bus. Okay. And so serialization means I should be able to do this in a scalable network. Whereas in a scalable network, I do not have the concept of one entity. There is no one entity which will be able to see what is happening throughout the network. Okay, because everything is distributed.

So here how do you do this? You can pause the video and think for yourself. How are you going to manage the serialization in such a big network? Right. So we do not have the bus which was there earlier, but we have a distributed system. In a distributed system, we have two options. One is that, do not cache the shared data and other is cache

the shared data. The easy option is do not cache the shared data.

What does this mean? All the shared variables will never be in the cache. They will always be only in the memory. Okay. So you have the home node. I am using the terminologies as discussed earlier. Home node has all the data and I say cache C1, C2, C3.

Whenever they need to change the shared variable X, they have to go to the home node because they cannot cache X. Okay. So the node C1, C2, C3 have to go to H to change X and the order in which they reach, maybe they will reach in this order, maybe they reach it in this order. Okay. So the order in which they reach H will be the serial order for that particular variable. And this will also guarantee that the reads, if there are reads in the sequence, they will always see the most recently written value to that particular location. Because everything is happening only at the home node in a first come first serve manner. Okay. Home is not going to shuffle them around.

It is going to serve it in a first come first serve manner. So home can be treated as the serializing entity, if I do not cache the shared data. But if I cache shared data, that is I have coherent caching, can home node be such an entity? If yes, should I do some changes or updations to make that happen? So that is my next question. Okay. So if I say home is my entity for serialization, one thing is that it should satisfy the request in the first come first served order. Yes, it should do it in FIFO order, so that I can determine some serialization.

But when I have multiple copies of a particular block, if the home sees it, home has seen that this changes has happened, it does not imply that all the nodes in the system have seen that change. That is at H, if you see that X has become 2, then X has become 4. Right. So I have generated this serialized order at H. But can you say that if H sees this, a remote node P1, somewhere else in the system, has P1 seen the same order? Can you guarantee this? May or may not be. Because the network is not going to guarantee a point to point transaction. The order in which these updates to X, which I am changing, H had seen X as 2 first and then it saw X as 4, it is not guaranteed that P1 will see it in the same order because they are separated by a huge network in the middle.

We are going to see counter examples to show that visibility at home does not imply visibility at the processes. For this, we will first try with an update based protocol and second with an invalidation based protocol. In an update based protocol, the network is scalable, so it does not preserve point to point order. The transactions can go in any order and reach at the endpoints at different time stamps. Here, the scenario is there are two write requests in two independent processes and they send the updates to the home

node.

So let me say N1 and N2 are my two nodes in the system. It is an update based protocol. So N1 is writing to the shared variable X and it makes X equal to 2 and it sends this information to H. Fair enough. N2 also writes to X and makes X equal to 4.

Both of them reach H in some order. Now let me say, the order is X becomes 2 and then X becomes 4 with respect to the home node. Okay. Let me say that H sees it in this order. Now H has to broadcast this update information to other sharers including N1 and N2 themselves and maybe N3 and N4. So I am not considering 1 and 2 but N3 and N4.

Suppose H has to transfer this information to N3 and N4, it will send first when it receives X equal to 2, it says okay, N3 take X equal to 2, N4 take X equal to 2. Then it sends X equal to 4, it sends X equal to 4. Right. So these messages that 2, 3 messages going, they may go in the network through various path might incur different amounts of network latency and so on. So eventually when they reach N3, N3 may see X as 2 first and then it sees X as 4. Whereas at N4, it is possible that X equal to 4 reaches first and then X equal to 2 reaches.

Okay. So with this what has happened, the order in which the updates were seen by N3 and N4 are different than the order which was seen by H. So these two are not equal. Okay. So H sees the same as N1, N3 whereas the order seen by home node is not equal to the order seen by N4. Hence even if I say, home could be my serializing entity, it would fail in this particular scenario. Next we'll take an invalidation based protocol and you would say that let me go back to strict request response, probably that works well.

So here I have again shown you the strict request response diagram where L sends a request to H, S sends a reply, then L contacts R and R sends this. In this scenario, now I have two other nodes which send a read exclusive request to the home node. We have H and we have two nodes L1 and L2. I'm taking two Ls because they are two new readers into the system.

Now both of them send a read X. Read X means they send a request that they want to modify the block. Both of them send read X to H. I suppose let me call this number 1 and when one reaches, H has to send a reply as message number 2 and in this what H is going to reply, it is going to send an acknowledgement and give the identity of the remote node so that L1 can contact the remote node to get the data block. Because that block is dirty in that R node. Let me draw this R.

So this R is a dirty node. It has to provide the data. Coming back to this, L2 sends a

read X. In response, H sends an acknowledgement and it says, okay, go to R and fetch the data. So it also gives the identity of R to L2. Now what do both L1 and L2 do? L1 and L2, both of them have got the identity of R.

So they go to R. Here L1 goes to R, L2 goes to R. Now what is the guarantee that both will go at the same time or in the proper order? If I list here with respect to the read portion, L1 came first and then L2 came with respect to the home node. So what will home remember? That L1 changed the data first, then L2 changed the data. Whereas when L1 and L2 go to R independently, suppose I label them, this has event 5 and 6, okay, transaction 5 and 6 can happen in any order and it is also possible that R will see L2 reaching it first and then L1 reaching it next. So again we have a mismatch, that home says L1 then L2, whereas dirty node says it was L2 then L1.

Even if we use a strict request-response protocol, here the home node and the remote node are still able to see different orders. Okay. So we have seen that even if the home can be made as a serializing entity, we have cases where there are scenarios that different nodes will still see a different order. That is the order in which the requesters reach the home node is not the same as it is seen by other nodes in the system. So which entity would provide a globally consistent serial order in this case? So what is the answer? Would you say home node or would you say the dirty node? And how to manage when multiple operations on the same block are in flight and serviced by different nodes? So there are some operations which the home is going to service, there are some which are being serviced by the dirty node, timing is not guaranteed, the order in which they reach one node is not the same as the order in which they reach other nodes. So there are a lot of complications in this, so how do I establish a serial order? So that is my question and we are going to answer this.

So what is the solution? I want a serializing single entity and my favorite is the home node. Can that home node help me? Well, let me add more power to this home node or let me put some more features in my interaction, so that I can manage this thing. Suppose a request reaches the home node and the home says that I will use additional states of busy or pending. That is the home is servicing a particular request and when it is servicing that request, it turns its state into a busy or a pending state. What does these two additional states help? They will say that some request is already in progress, it is not completed and hence we can say that let me postpone further request or take appropriate actions on newer requests.

So when new requests arrive, the serialization can be provided by some number of methods or I would say the serialization can be done by multiple entities in the system and not only by the home node. So let us see the different mechanisms. So we can say

that let the home node be the serializing entity. If it has to serialize, it has to make sure that it handles one transaction at a time and one by one. Okay. So if I am giving the responsibility to the home node, home node keeps track of the transaction in the particular order they arrive and it serves them one after the other and never services the next transaction before the first one has finished.

So that is using buffering at the home node. The second option is, I could buffer at the requester. That is the requester has sent. Let the requester keep the responsibility and not bother the home node. We have to somehow manage that multiple requesters will still be able to remember that they had sent the request and maintain the order of the requests.

The third is NACK on retry. I would say the easiest one. You just simply send a negative acknowledgement and let the requester try it out in future. And the fourth is forward to the dirty node, because the dirty node has the data with it. Let it handle multiple requests and let us release the home node from the responsibility. Okay, so these are the four possible cases and we are going to do them one by one.

Buffer at home. So new request is buffered at the home until the previous request in progress has completed. So this is my objective. So let us say L1 was a first requester. It went to the home node with its request and this request was forwarded to R and it is still yet to get a response. So home says that I am busy right now for that particular data block.

It is not busy for all blocks, but with respect to this particular block, say A, it is handling. It has to keep servicing request to other blocks. Otherwise we will have protocol level deadlocks. So H says I am busy with respect to block A. Now suppose L2, L3 and L4, these are four, three , four more requesters, they all go to home node for the same block.

Now what should home node do? Home says that I am busy with A, but okay, you all wait in the queue. So it puts them in the buffer here. So this is called buffer at home. So it puts these requests in the buffer, first L2, L3 and then L4. So even if the request is being forwarded to the owner, that is the owner R is going to do the work, it will eventually get a response, but future requests will be put in a FIFO order at the home node.

So that was buffer at home. Now any problems with this? When I buffer things, I have problem of storage, that is buffer overflow may happen if the home does not have sufficient capacity to hold all the pending requests. You would say, well can I use the memory close by? Okay, possible tha,t if your buffer space falls short, you can overflow

your newer request to the local memory node and then use them in future, that is possible. Then it reduces concurrency because the L1, L2, L3, they are all queued up, so I am not permitted to do transactions in parallel. So that flexibility reduces and hence we have lesser concurrency in this particular solution. And here, because the home has declared itself busy, whenever R finishes or L1 has finally got the data item, they both have to somehow guarantee that H knows that it no longer remain busy.

So H has to come out of the busy state and it needs to be informed by the nodes which are involved. Okay. So we need to notify the home node when its involvement is over. This additional feature has to be guaranteed by the protocol. Okay. So buffer at home is a viable solution, only thing we need to guarantee these points 1, 2 and 3 and this is used in the real life processor, MIT-Alewife processor uses this particular optimization and solution.

Next is buffer at the requester. So pending requests are buffered at the requester and not at the home node. Now what is the good thing here? If at the home node, they will be for the same block and there will be several such, so there will be capacity issues, buffer capacity issue at the home. Here I am putting the responsibility with the requester. Now if you can imagine this is going to work very well in a distributed linkless type of a design that is the cache based directory. Okay. Let us see how we have L1 which sends a request to the home node which is getting serviced by R.Okay. 1, 2, the 3 has not yet come, home is busy.

Now L2 comes, L2 goes to the home node, home says I am not going to buffer you, I am not going to serve you and I am not going to reject your request, but you queue up somewhere else, don't queue up at my node. So L2 says where do I queue up, who are the other people waiting the queue? So it says okay, I know that L1 is being serviced right now, maybe you can stand behind L1. So when this one goes to H, H will reply to L2 back, so this is a separate transaction and then L2 lines up with L1 or rather it builds a distributed linked list with L1 and L2. So this is not actually going and waiting in the buffer at L1, but we establish a pointer between the two requesters and this way we can have a link list done. Suppose L3 comes, again it goes to H, H says alright, go and join the queue. So instead of L1 now H gives the identity of L2 and here these two get linked. Okay.

So this way I am going to buffer the, buffer the pending request at the requesters and not at the home node. So the request is still sitting at L2, the request hasn't reached H, it hasn't reached R, it is still sitting in its own node, but only having a pointer in the queue. So once L1 has finished, L2 will get its chance to get served. Okay. Now the third one, NACK on retry. So here the home need not buffer the request, it simply sends a negative

acknowledgement to the requester.

So we have the same scenario and home is busy. When L2 sends a request to H, H sends NACK. Okay. I cannot serve your request, retry later. So L2, L3, whoever come in future, will have to do the retry of the request in future. So this idea is implemented in the SGI Origin 2000 protocol processor. Now what about the order of serialization here? So would you say that the serial order was L1, L2, L3 or which one? So I will say definitely L1 because L1 was getting serviced but the order of NACKs don't decide the serial order.

The order of getting serviced decides it. So suppose L3 wins, probably then L3 happens first and then L2 happens. Because the retries when they succeed will decide the order of serialization. Fourth one, forward to the dirty node. Let dirty node take the responsibility. Subsequent request will not be buffered, neither NACKed, but they will all pile up at the H, at the R node. Okay. Because home was busy, what happens? When L2 comes to H, H says okay, go and wait at R, because R is going to service you.

So overall at R for a particular data block, you are going to get a queue of pending requests. So this is forwarding the request to the dirty node. Dirty node is going to buffer them and the order in which they reach the dirty node is the order of serialization. Suppose L4 came and L4 reached first, probably L4 reached before L2 then L4, L2, L3 would be the order. So order of serialization is the order in which the request reached the dirty node.

And if there was no dirty node in the system, what would you do? Then they will have to be buffered at the home node, because there is no dirty node, home will service them one by one. So here the order of serialization will be determined by the home node, again in the first come first served order for clean blocks. So if the block is clean, order is determined by home. If the block is dirty, order is determined by the dirty node.

So far so good. But now in the meanwhile suppose the dirty node deletes the block and we already have the queue L2, L4, L3 lined up at R, what do we do? So when R, in the meanwhile, suppose R deletes the data block with it. So in the meanwhile, this queue is being formed and here the block gets deleted. Now what happens? So this queue will have to be handled. So essentially R is going to look up this L4, L3, all these requests and it is simply going to send a NACK to them.

That is I cannot serve you any further because I don't have the data block. Once the request is NACK'ed it will be then retried later in future. So in the meanwhile, if the dirty node release the data block, that is it writes back this data then it is going to send

NACKs to all the new requests. These will be later retried and whenever they are retried later, that time the serial order will be then decided either by the home, if the block is clean or by the dirty node, if the block is dirty. This idea is implemented in the Stanford DASH protocol processor. Fine. So we have tried to establish a serialization order. But is a single entity still sufficient to establish the serialization when I have multiple copies that are distributed throughout the network.

The problem I am trying to highlight here is that the home or the serializing agent, be the dirty node or the linked list and so on. So they may see the order, they may know that their involvement is done or not with respect to a particular request, but if the serializing entity sees that yes, I have finished the request in this particular order, does it mean that the request has finished with respect to all other processors? Have all other processors seen that transaction being complete? Because there may be some transactions for a next request to that same block which may go to some other nodes and get serviced through those nodes, in parallel to this transaction. So I have one transaction going on. In the meantime another transaction can start and finish with respect to this node, with respect to this location but some other set of nodes.

So this can still happen. Now you would wonder how but we actually do specific solutions for this when we handle the case studies. Right now nothing to do here. But again the point to take home is, that single serializing entity is not sufficient. In addition to having such an entity, I still need to guarantee that local serialization at every node should be done. That is each node which is sending a transaction should not serve future transactions until its pending transaction is complete. So suppose N1 is a node which is having a pending transaction on block A and suppose another new transaction comes for the same block A from elsewhere to this node.

So N1 should guarantee that because already one transaction is in process, it should not serve new incoming transactions. Okay. So this is how we will be able to guarantee local serialization. So we have a global serialization entity, which is the home node or the dirty node depending on those four options. And apart from the global serializing entity each node locally has to also maintain a serializing order. That is, they should not serve incoming transactions or serving transaction means if an update comes to this node when it is already sending an update, so it should wait until its own work is done before applying the newer transactions.

So we have seen the concept of serializing order with respect to a given location. We will do more correctness aspects in the next lecture. Thank you so much.