Parallel Computer Architecture

Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati

Week - 08
Lecture - 42

Lec 42: Directory Organisations

Hello everyone, we are doing module 5 on Scalable Shared Memory Systems. This is lecture number 3 on directory organization. We are going to see the different types of organizations in which we can store the directory information and we will go through this chart in brief, that we have several directory schemes they can be either centralized or distributed that is the directory information about what are the different sharers of a given block can be kept in one centralized location, that is the centralized type or it would be distributed. And when we have a distributed organization, we have more flavors available, one is called a flat organization and the second is called a hierarchical organization. Hierarchical as the word says, it is like a tree structure and flat, it is a flat structure. So there is no hierarchy or no parent or child nodes in this hierarchy. So we just have a flat organization. In that we have again two categories, one is a memory based and one is a cache based. So these are the four types in which the directory structure will get stored.

So we will start with one by one, understanding them, and definitely we are going to do in detail the last two, that is the memory based and the cache based. But in this lecture, we are going to see an overview of all these types. Right. So directory types centralized is the first one. As we understand the word centralized is just one location in which information of all the blocks throughout the whole system is kept. Right. So if you have 10 nodes, each node having one memory bank and several blocks in that memory, all those information is kept in a single location in a centralized place and all the sharers of these particular memory blocks is kept in one centralized location.

Now this storage, however optimized data structure if we draw, it is going to have a limitation on how many nodes it can cater to. For example I designed a system with say 100 nodes. Tomorrow I add some more nodes to my system so my centralized storage has to again scale up. I have to change the storage style, so that I can accommodate the new 100 nodes which I am adding. So it is not a static structure. I need to keep on updating it. However the advantage of such a system is, I have a one point contact, that I go to one place and I get information of any block I wish to obtain. So that is the centralized one.

In distributed, as the word says that information is distributed, it is distributed across the nodes and especially it is associated with the memory. Because every node is kind of associated with a memory bank and this particular memory bank can house the information about the sharers, that is the memory based flat structure. Then we have a cache based design where the individual caches can store the information and the third type of distribution is hierarchical. So we have centralized where we have the disadvantage that if we keep on adding more and more processors, the structure will keep changing and we need to have more scalable design to emerge. Right. So as the number of nodes increases, your directory structure is going to change. The advantage is that, you have one point contact to go and get the information and we can send update or invalidation messages only to those nodes that have a copy. Right.

What does this help me in? This helps me in doing a no broadcast. If you recollect snooping, when I adapt it to a bigger system, we end up doing a global broadcast of the snoop messages but here we can avoid broadcast. In the distributed, we have two types memory based, cache based and the third one is hierarchical. So how do I find the information about the sharers? In centralized, it is very easy just one place. But the disadvantage of centralized is that, it is not scalable. But the implementation is easy, so that I can say this is a positive aspect. In distributed, we have two structures ,the flat and the hierarchical.

In flat the directory information is with the home nodes. So you need to know where the home node is and I think that is very easy to find out using the address of the cache block. So once you know the address of the cache block or the memory block, you can simply use the hashing to find out the home node address and we can directly send a network transaction to the home node. So once you know the address, you know the home node, you go to the home, and find out the information of the sharers. In the distributed hierarchical, however there is a different structure. Because here we are kind of building a tree of information.

So I will have leaves of processor nodes. I will have an internal directory, so it is, although it is distributed but it is not a flat directory. When I say flat I have just one location like this or a set of locations or bits and pieces of information which is in one dimension. Whereas when I say hierarchical, we have a tree type of a structure in which that information has to be gathered and identified. Okay. So we will see a little bit more on hierarchical. Right. So here the leaves are the processor nodes.

I will just take a hypothetical system where we have a scalable network connected to say several nodes and each node can have some more processors connected. I am taking two levels here but you can have multiple. For example, this is connected to another network which is then connected to more processors and so on. Okay. So you can imagine a big system like this and I want to maintain information about the sharers in the directory. Okay. So these processor nodes, the small circles are the processor nodes, they will be the leaf nodes of my hierarchy. So I will say that P1, P2, P3, these are my leaf nodes at which the information about their local sharers or the cache state is kept.

Now there is a parent node to this which keeps information about the sharers within its sub tree. And now this node which I am calling a parent of P1, P2, P3, it could be the node P1 also, because P1 suppose I say, this is my P1, P1 can keep information about itself and its child nodes. Okay. So we do not have special nodes called these intermediate nodes. One of these processor nodes itself is going to act as an intermediate node. So an intermediate node also is one of the sharers.

In addition it also maintains information about its child nodes. So it will maintain information about what P2, P3, P4 is keeping, including what it is housing. Right. So this is the way in which a logical hierarchy is constructed. If you can quickly see that this P1, P2, P3 is not physically looking like a tree in my diagram. The P1, P2, P3 are actually symmetrically distributed in the interconnection network , but they look like a logical tree in my hierarchical directory.

So these intermediate nodes are going to keep information of the child nodes. Right. So every processing node not only serves as a leaf node, like here we have P1, P1 is serving as a leaf node but at the same time it contains the directory information of all the internal nodes that is it also contains information of its child nodes as well as its own nodes. So moving or zooming into this more, if I see how the hierarchical directories work , now that you have got an idea how the leaf nodes look like, I have drawn the tree upside down, the leaf nodes are at the top. Now these are the processing nodes, they have their own caches and cache state information. This is the level 1 directory.

Now this level 1 directory is, it is not a separate node, it is one of these nodes which keeps information of itself and its child nodes. So what does this intermediate directory, level 1 directory do? It keeps track of its, of its child processing nodes that have a copy of the memory blocks. Okay. So it is going to know what all blocks are cached by its child processors within the hierarchy. They are not its child processors physically, but level 1 directory the node which is housing it has to keep information about the sharing status of its child nodes. In addition, it also keeps track of its local memory blocks which are housed outside. Right.

So if there is a block A which is local block to this processor and suppose this block A is in the cache of this particular processor. So this level 1 directory should know that the block A which is with me is now gone outside to some other node. So it has to keep track of which blocks, which local blocks have gone out and it also has to keep track of which remote blocks have come in. Right. So this B has come in, so it has to know that B is a remote block, A is my own block which has gone out. So all this information it has to maintain.

So that is level 1. Level 2 again you can scale that problem further. Level 2 maintains information about the level 1 directories and this is how the scalability is achieved because level level 2 does not need to know about every individual processor node but only needs to know about level 1 directories. So it also tracks which local memory blocks are cached outside its sub tree. Now we are talking about sub tree. Now for this one, A and B are local nodes. Right.

So they are its local nodes because they belong to its sub tree and there could be a node C which comes from outside elsewhere from another branch of the tree. Okay. So this is how a hierarchical directory structure is constructed. This is a logical directory structure and how will you search a particular block or information about it. So we need to ,when a processor sends a request, that request percolates up through the hierarchy from that processor controller to its parent. Suppose this node wants some information.

It asks its cache controller. It might go here. If this is able to satisfy it within its sub tree, it is okay. Otherwise it goes out to the next level.

Then level 2 again same. If it can satisfy within its sub tree it is good. Otherwise it might go out to the next level. So the search message percolates from the leaf nodes up to the root and possibly on the way it is able to gather the complete information. And hence we have point to point messages sent from child to parent nodes.

Right. So we have seen the structure of a hierarchical directory and here the directory information is kept as a presence bit vector with every intermediate node in that complete tree and real life implementation of the data diffusion machine which implements such a hierarchical directory structure. The next distributed structure was the flat directory. Here, the information about sharers is kept with each individual memory bank and essentially with every block of the memory. That is, if the memory has got 1000 blocks, it has got 1000 directory entries, each keeping information of which all nodes are sharing this particular data block. Now this memory based flat distributed structure, here, it is implemented by the Stanford Dash processor, the MIT Alewife and the SGI origin systems.

So these are 3 to 4 real life implementations which use the flat memory based structure. The other variant of flat is the cache based where instead of the memory keeping the information, every individual cache keeps the information about the sharers. That is, a block in the cache has to know that it is being shared by which other nodes. So essentially we establish a kind of a linked list between cache blocks across the whole system which keep, so this list is going to keep track of the different sharers. So this cache based information is kind of distributed because the blocks are in different nodes.

We establish a link between these blocks to know which are the different sharers and the implementations are the SCI IEEE standard implements the cache based and the sequent NUMA-Q architecture. So among these few examples we are going to see the cache based distributed directory, how it works using the case study of the sequent NUMA-Q and we are going to see the memory based using the SGI origin protocol. Okay. So you will get more understanding about the working when we do these two case studies. Okay. Continuing with the overview, the flat memory based scheme, if you see the picture on the right hand side this, we had discussed in the previous lecture where you have these nodes , the interconnection network ,the memory associated with the directory ,the flat memory based. So this is flat memory based means directory is associated with the memory. Okay.

How does it perform, if I ask you how much traffic does it generate on a write. So you can pause the video and try to answer these questions for yourself , that is how much traffic does this generate, how much latency overhead is there, and what is the storage overhead for this particular design. Okay. So if I am talking about traffic on a write, what happens on a writ,e I have to send invalidations to all the sharers, so how much traffic am I generating is it in the critical path. Suppose I have 5 sharers and later suppose I have 10 sharers. Is the 10 sharers traffic going to be more? is the 10 sharers latency going to be more? So these questions we have to answer. So traffic is definitely going to be more for more sharers. But we can send the invalidation or update messages in parallel. So the latency is not going to scale with the number of sharers. Okay. So the critical path, that is the longest number of transactions before I can start writing, is not proportional to the number of sharers. So that is the good point about flat memory based system. Traffic yes, it scales with the number of sharers. Then the storage overhead, if you can see we have a bit vector and the bit vector size is dependent on how many nodes I have in the system. I have 100 nodes, so this bit vector will have 100 bits, if I have 1000 nodes then I have to scale the same bit vector to 1000 bits. So is it practical?

So well of course you would think that may not be but then how do I solve this problem if I still want to stick to this architecture. So we are going to see different methods of optimizing this bit vector storage to make it more scalable. All right. So that was flat memory based scheme. In this architecture, how do you do a read and write? We had done a quick recap of this in the previous lecture. If you recollect, in this architecture when a read happens, the processors doing the read goes to the directory and the directory will put that particular bit to on, that is if 10th processor of P[j] is asking for the information, so P[j] bit will become 1 in this bit vector and the block will be sent to the reader.

This is the read process. What happens on the write? On a write you have to send invalidations to all the existing sharers. So you know the sharers from this list, you will know what are the different sharers and you will send invalidations to all of them. Then you will remove all these bits from the sharers list because they no longer have the block. Later, the newest writer will be made on in this list and you have to turn the dirty bit on. Okay. So if this was too fast you will get more feel of this when we do the case study. But right now the overview is, when a write happens, send invalidations to others and set the dirty bit on and give the block to the new writer. Okay. Now we will move on to the flat cache based schemes. So if you see this slide here, you have the main memory, which is the home node ,and I am talking about suppose this data block.

Now this data block does not have a directory associated with its entry. I will go back to the previous slide for you to compare. This was the memory and with the memory you had the bit vector telling the information about the sharers. Now this directory is not with the memory. Now where do I keep this information? All these presence bits. Right. All these dots where do I keep them? Right. So they are kept with the individual caches.

So this home node is going to have a pointer to the first sharer or the head node. Okay. So I will construct a linked list and the head node happens to be now the first sharer right now. So node 0 is the head node. So this particular line in the cache of node 0 is housing this block. Suppose I say this is block A, this block A is here, it is here and there. So now node 0 knows that it is housing this block which is the other sharer.

So node 0 is now going to point to the next sharer which is node 1. Then node 1 is going to point to the next sharer which is node 2 and so on. So this list can continue to the number of sharers. If there are no sharers after this then you point it to null. And of course for quicker optimized implementations, most of the time we have a doubly linked list in this case. Okay. So the memory does not store the information, it only stores pointer to the head node.

The list is maintained in a distributed fashion, right. Linked list is distributed because these node 0, node 1 and node 2 are at these joint positions in your complete system. So they are at distributed locations and they point to each other forming a distributed doubly linked list. Each of these caches, what does it have? It has its own tag, its coherence state which we normally need for coherence protocols but additionally it needs now two pointers, the forward and the backward pointer to maintain this linked list. How does read and write happen in the cache based system? So how does read and write happen when I am using a flat cache based system? When a read is issued by a processor we have to go to the home node using the address.

So we first go to the home node and the home node does not keep the bit vector like in a flat memory based system but the home node only keeps a pointer to the sharers list, that is the head pointer. Okay. So suppose this is, the purple one is my home node, that is the memory it has the data and with every data block it houses the pointer to the home node. So here it says that the data block A is shared by nobody because the pointer is null. So if the head pointer is null, then there are no sharers, it is a read request.

So this data block is given to the requester. So you give this data block to the requester and now you know that the requester is now one of the sharers. So if I say the requester was suppose sorry P4.If the requester was P4 then I now need to point this to P4. So this way we have started constructing the linked list. And this P4, if I elaborate further, you have a cache which is associated with the node P4, this is the block A and this will now point its next pointer will be null because it does not have the next sharer its previous pointer will go to the where will this go? It will go to the home node. Okay. So this is how I am going to start constructing the linked list. So home, points to the new requester, new requester's back pointer comes back to the home node and here data is provided by the home node.

Now suppose the head pointer was not null. In the same example, now P4 wants the data but it comes to the home node and sees that P3 is already having the data copy. So what should home do? One thing is either the data is fresh with the home node or possibly P3 has modified it. So there are protocol methods which will help us to identify this. But for the time being we can make a generic statement that, either the home node or the head node is going to give the data. So data will be provided by the home node, if it has the latest copy or by the head node, in case the head has modified the data copy. So what does it do? It says that P4 has come for the data block.

The home node tells P4 that you go to P3 and add yourself to the linked list. Because even if home node points to P4, P4 has to attach itself to the head of the list. So home gives the information about P3 to P4. So P4 goes to P3 and establishes the link here. So this is going to be my new head node.

So P4 becomes the new head. P4's next node becomes the old head and now P3, P3's back pointer was initially going to the home node but we disconnect this and now connect it to P4. Okay.So this way the new requester always gets added to the head node and you would understand why this is done. Because this reduces the number of messages or complexity of the protocol. Suppose I had 100 nodes, I am not going to traverse the complete list of 100 nodes to attach myself at the tail. It is always quicker to attach at the head because the home node keeps information about the head node.

So that is about read. What happens on the write? So here I have constructed a scenario where P3 is the head node. So this is node P3 and then there are a few more nodes, node B, node C, node C. So this should be node P3. Okay.So now a write has come and if there were no sharers, that is the head pointer was null, the data could be given and the new node becomes the head node and they can start writing. But suppose there were already existing sharers like these A, B, C or P3, B and C.

So there are these three nodes already sharing the data. So we need to invalidate them. So how do I invalidate? So I need to traverse the complete list, sending an invalidation message in this order. So you send the invalidation from here to here, here to here, here to here and now C will then send an acknowledgement that yes, I have invalidated ACK returns and other ACK returns. Once everybody has finished invalidation, then , thenP3 is now going to point to the new node and this complete list from here to here gets purged. Right. The nodes head node, node B, node C will be removed from the list and only the new writer will get added to the list.

So okay. Okay. So we have seen the write. Now what happens on a write back or a replacement? That is, one cache deletes this particular block and when it deletes the block, it is aware that it is part of this global distributed linked list and it has to remove itself from the list. There is no home node or one parent node to whom it can just tell that remove me. It has to take voluntary actions to remove itself from the linked list. So what should it do? It has to tell its previous and the next node to connect to each other so that it can be associated. For example, if you have A, B and C connected and B wants to remove, so now A has to connect to C and C has to connect back to A so that the linked list is established and the node B can be removed, because the node B has deleted the block and it no longer is one of the sharers.

Now while doing this, there are possible race conditions or synchronization issues that when B removes itself, it tells A to connect to C but it may happen that maybe A is also wanting to remove itself or probably C is wanting to remove itself and there are 3 to 4 messages which will be exchanged between these nodes, and possibly within that duration, the nodes A and C also might take their independent decisions to delete or update themselves. Okay. So we need to take care of synchronization issues in this particular case. So this particular cache based directory structure is implemented by the scalable cache coherence interface, that is the IEEE design standard for scalable coherence. So on a write back, we have to delete, so this node has to delete, finally making A and C connected, we have to handle synchronization, and this is implemented in the SCI IEEE standard. Okay. So what are the disadvantages of a cache based system? Again you can pause the video and try to answer questions like what's the latency on a write, what is the traffic light and so on. Okay.

So what is the latency on a write? When we have to write, we have to invalidate all the sharers. Compare this with the flat memory based system, where in a flat system you had to send invalidations but you could do it in parallel because you had the complete list with you. So you could do it in parallel whereas here, you don't have the complete list, only you have list of the next node or after you reach to the next node, you know the next node next to it and so on. So the traffic or the latency on a write is proportional to the number of sharers, so it's going to take a long time. Although the traffic could be more both in the flat and the cache based, the latency in a cache based is much more than the flat memory based system. The other disadvantage is every communication assist with every node, so when you start invalidating or talking to every node in the system, you are disturbing that node, you are going through its communication assist and telling that now go up to the cache, invalidate, come back.So we are disturbing all these communication assists on the way which is going to involve more energy, power and also add to the delay because the assist will take time to go up and invalidate and then respond back to us.

In case it is busy doing other tasks, again more waiting time. So overall cache based systems will add to your latency. So that was in case of writes but in case of reads, is it fast? Well, not as bad as write. But still when you do a read, what are you doing? You have to go to the home node, then attach yourself with the previous head and the previous head attaches itself to the new requester and back. Okay. So overall three nodes get involved even for a read node. So latency wise cache based is going to take time but it has advantages.

What are the advantages? Storage and fairness. Storage overhead is not so much because every memory is going to only store a pointer to the head node. Right. The memory block, what is the overhead? We do not have the bit vector like we had in the flat structure. So this bit vector is not there. We only have a single pointer. So overhead is less and with the cache blocks, we have to add little more overhead with only two pointers.

So every cache block, suppose block A, it has to store pointer to this next node and a backward pointer. So forward and the backward pointers are to be kept, which adds to the overhead but still not so much like a flat bit vector style. And we of course have fairness in the system. Fairness because once I construct a linked list, I know the order of the requesters and I can be fair saying that, I first serve this one because it had come to begin with and then first one came, then two came, then three came. So I can establish an order of accesses and then give fair accesses and fair responses to the requester. And it also avoids live lock.

So in terms of fairness and storage overhead, cache based systems are good. The other advantage is, it being distributed, we do not have to work in a centralized manner. So the work is not centralized. We are going to spread out the actions to be taken. Although we involve the communication assists at every node, but still it reduces the overall bandwidth demand on the system. Because at the home node, if home node had to send all the invalidations out like this, then wait for all the acknowledgments coming in, coming in. So your bandwidth and contention at the home node will go up in a flat memory based system, whereas in a flat cache based system, your bandwidth demand at the home node is reduced.

And what happens on traffic on a write? Traffic on a write, similar to a flat based system, it is proportional to the number of sharers because traffic is not going to reduce. Only thing is the latency is high in a cache based system. So this was an overview of the different types of directory organizations. With this, we complete this lecture. Thank you so much.