# Parallel Computing

**Prof. Subodh Kumar**
**Department of Computer Science & Engineering**
**Indian Institute of Technology – Delhi**

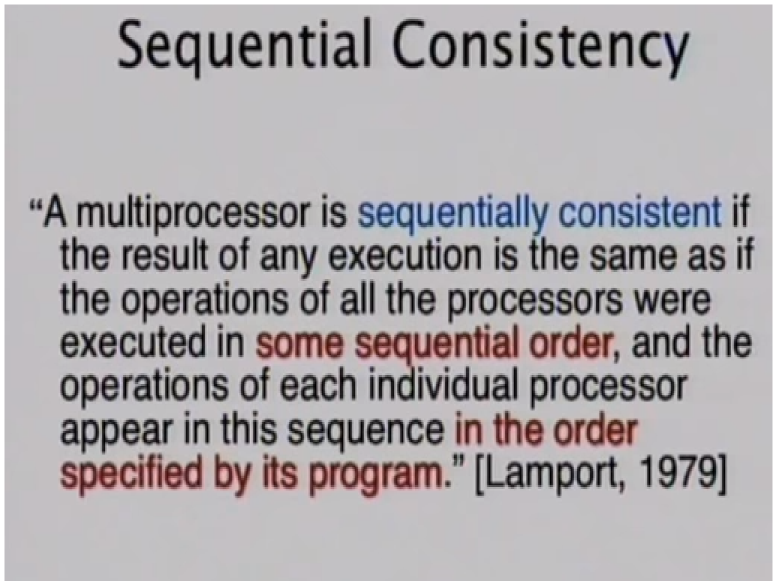
**Module No # 03**
**Lecture No # 12**
## Memory Consistency & Performance Issues

Alright so now we are going to get back to what we were discussing if you remember we were talking about memory consistency models in the context of multi-processing and some of those also applied to database because there is also same issues come and there is concurrency there are multiple transaction in that context associated with the same piece of data.

And ultimately it has to be some expectation that the programmer makes from how the data gets modified. And we are going to limit ourselves to multi-processing where you have not talking about last transaction but small access which is read write into a some location that may be directly a shared memory location or something that appears like a shared memory something that is accessible a memory location that is remote.

We discussed strict consistency where you basically have everything appear instantly there is a global clock we make a read and instantly they have returned the things that read you make a write and instantly it is written okay.

**(Refer Slide Time: 01:55)**



## Sequential Consistency

"A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

In sequential consistency we do recognize that will not happen constantly you will have something that begins and sometime later it will end so read start of read will make that request end of the read will be the data comes back. Similarly sort of write is when you make the request end of the write it is when everybody sees that new data okay beyond that pass that point. And so this is a notion of sequential consistency says that there are this read and write accesses to shared memory they can be ordered as if they were being run in a sequential environment right.

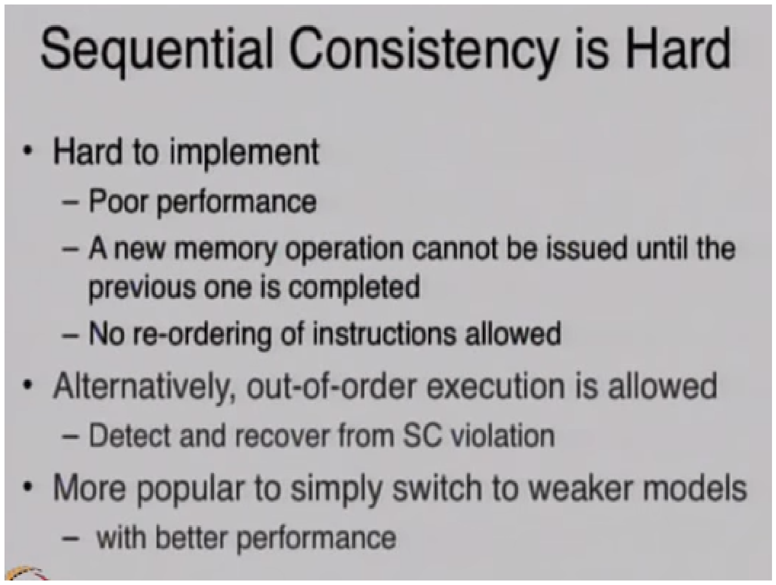
Meaning that all the different reads and writes of multiple processors have to be put in one linear order. In this order so this order must be what the program ultimately saw must be consistent with the program overall program the multi processors parallel programs are and it must be consistent of each processors sort okay. So if a processor says read then write then that is what the overall global order contains.

There was also variance order contains not of sequential order consistency but mostly of strict consistency which is linear where it was perfectly instantaneous said that you say read and read happens at that exact moment. There is some lean way some laxity and says I said read here my response came back there and sometime during that period there is as if that read happens instantly during at that spot between my sending of the request and getting the response back similarly for the right.

There also the reason notion of the global clock everybody is here is the read happening at this time here is the read happening at this time and then their consistency order with respect to each other if they are not overlap okay. And even they are overlapping there will be some order but that order is not fixed because this read can be atomized at any point during that process during the getting sending the request and the response right processors okay.

Alright so I also mentioned that sequential consistency is somewhat order to implement it is already somewhat looser than strict consistency.

**(Refer Slide Time: 04:43)**

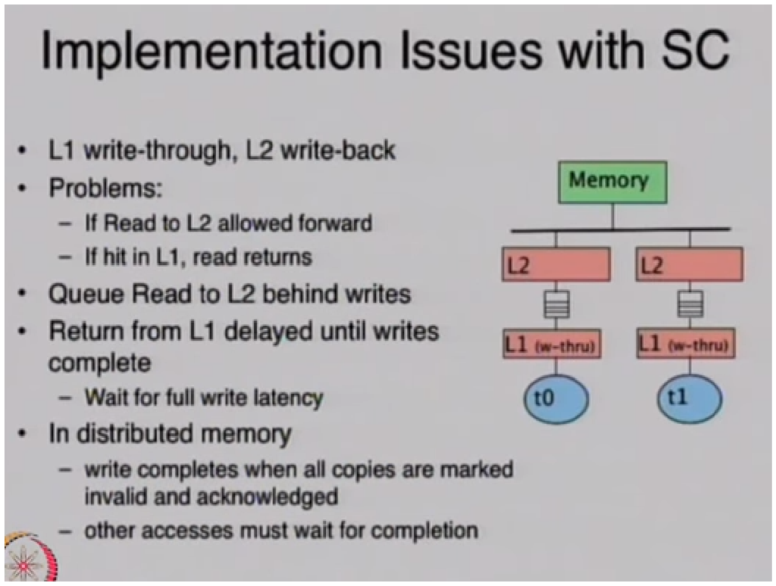


## Sequential Consistency is Hard

- Hard to implement
  - Poor performance
  - A new memory operation cannot be issued until the previous one is completed
  - No re-ordering of instructions allowed
- Alternatively, out-of-order execution is allowed
  - Detect and recover from SC violation
- More popular to simply switch to weaker models
  - with better performance

And the reasons that it is hard to implement is essentially that it is saying that everything that is in the program order must be maintained. If the program said write then read then in your final response every other processor must see that write then that read okay. And that means that there is not reorder is allowed in a complier cannot reorder and transactions to memory read and write transactions to memory cannot pass each other okay.

**(Refer Slide Time: 05:27)**



## Implementation Issues with SC

- L1 write-through, L2 write-back
- Problems:
  - If Read to L2 allowed forward
  - If hit in L1, read returns
- Queue Read to L2 behind writes
- Return from L1 delayed until writes complete
  - Wait for full write latency
- In distributed memory
  - write completes when all copies are marked invalid and acknowledged
  - other accesses must wait for completion

Memory
L2
L2
L1 (w-thru)
L1 (w-thru)
t0
t1

And it is actually quite easy for them to pass in fact most architectures allow them to pass and that is why and that the reason they allow them to pass is to keep things efficient and keep it not keep not every operations strut behind a long operation like a read like a write which will typically take longer than read. So here is an example we have got 2 processors marked in T0 and T1 in our terminology will be calling them threads they may be running on the same processors they may be running on multiple processors different course on the same processors whatever.

So they are this threads that are that had local L1 cache okay and L1 cache is backed up by L2 cache L1 cache is write through cache is right back and then there is memory okay and the data when you send when you write something to the memory it is going to check if it is cache right and it is right through so it has that memory right has to go through all the way to the main memory okay.

So that memory write transaction will get into a buffer between L1 an L2 in this case it was stopping but L2 write through will also have a similar issue. Write through means whatever you write has to also with written into the back store into whatever this thing is presenting and right back is lazy am going to write it here later on I can read directly from here but the right back will happen when this needs to be active or some other condition becomes true alright.

So you when there is a single processor then there is no problem because what is going to happen is the writes will gets queued in the memory in this buffer and it is possible that you read something can that read effectively finish before that write even in one processor it can so there are reads and writes that can processor is showing and there is an L1 cache if you write then the write transaction has to go through right.

If you read and if it is L1 cache then it is going to get return otherwise if it does not get written then this read is waiting for that write okay. But what is going to happen in single processor it is possible that read written from cache and the write is not still not done it is in queue. However if it is non-conflicting read mean that reading something else we have written something else then you do not care.

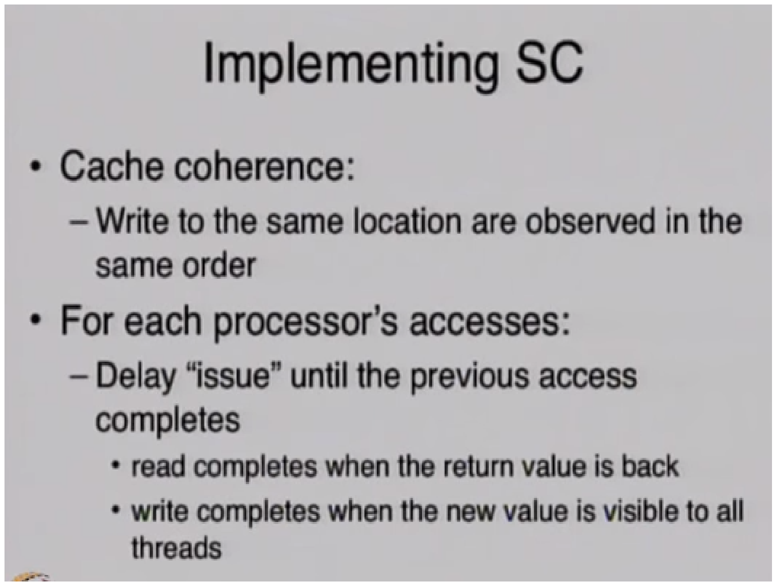
If it is a conflicting read you are reading the same thing that you are written your cache is up to date you have written it other one before sending it down okay. So that re-ordering can happen and allowed but nobody care out now you are saying because every other processor must also see which order you would be showing transactions I cannot have you see a different order and other processor see a different order.

As a result of which the read cannot bypass the write even if it is unrelated so you wrote to one variable it is struck in the queue then you read something else sitting (()) (09:53) in the cache but you cannot return it because otherwise other people will see that the read later but you will see the read happen first it came back first you read from the cache. But the read transaction will be ultimately serialized right it has to be put somewhere and so if other people do not see your read happen first then they may behave as if your read happen later.

As a result of which you cannot put this one wants read here the other people want to read there and so there is no way to serialize and so in maintaining sequential consistency essentially every reordering has to be killed everything as to be in order and not just in order but we have to wait for the previous transaction to finish before this transaction can proceed. As a result of which you are essentially paying for the latency for every single time there is not benefit of cache unless you are doing only reads in which case it is perfectly fine.

And this queue can get even worst if this is not just going to the same shared memory of to a common bus which keep this ordered but on to a network where some shared memory on this machine some shared memory or some access memory is on this machine some on that machine some on third machine it only get worse okay so it an latency will get worse which means everybody have to pay the price for every latency okay.

**(Refer Slide Time: 12:00)**

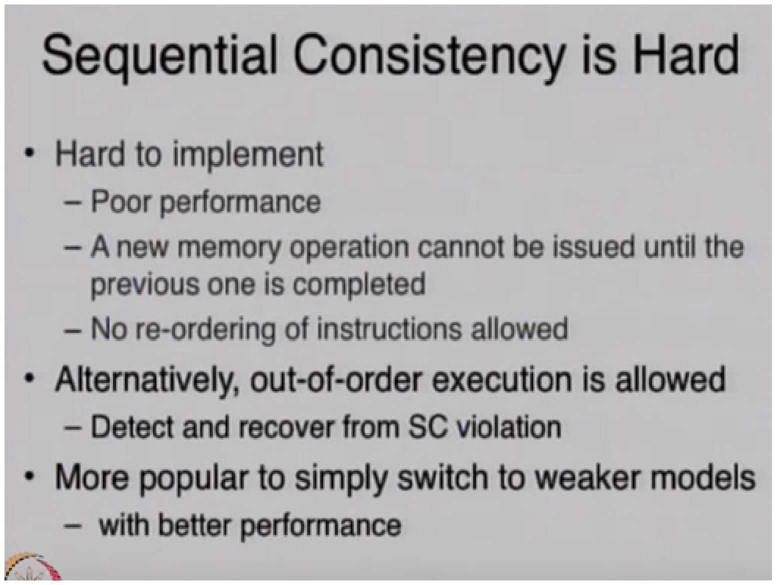


## Implementing SC

- Cache coherence:
  - Write to the same location are observed in the same order
- For each processor's accesses:
  - Delay "issue" until the previous access completes
    - read completes when the return value is back
    - write completes when the new value is visible to all threads

So essentially in order to implement sequential consistency we are going to have to maintain this ordering and the only way guarantee this is saying do not issue the next until the pervious one is completed. You completed means visible to everybody thinks it as completed okay. So if you are sending it to a remote location you must get and act back that it has completed before you can know the way it is completed and that act is being sent to everybody.

**(Refer Slide Time: 12:00)**



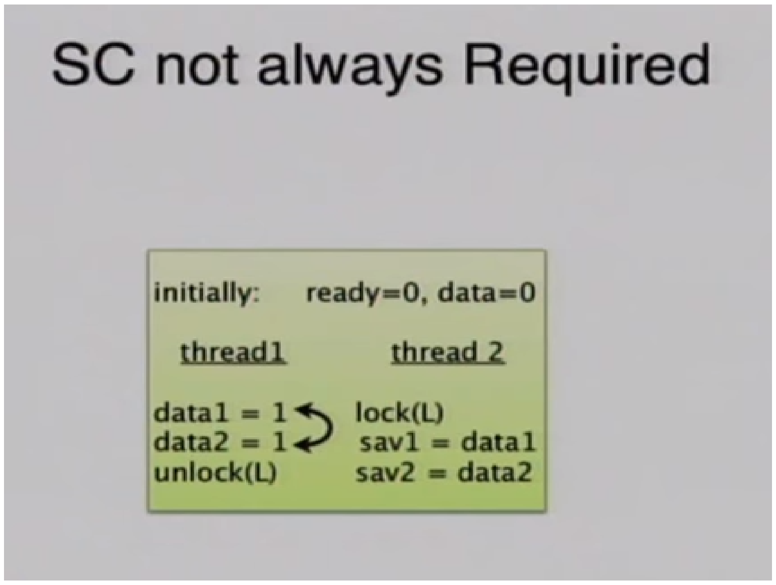
## Sequential Consistency is Hard

- Hard to implement
  - Poor performance
  - A new memory operation cannot be issued until the previous one is completed
  - No re-ordering of instructions allowed
- Alternatively, out-of-order execution is allowed
  - Detect and recover from SC violation
- More popular to simply switch to weaker models
  - with better performance

So there are variance one is you last thing to proceed out of order but you keep detecting if later on there is somebody else that might see it in the wrong order okay. And this get quite complicated quite quickly where you allowed it to get out of order but keep track of what out of order things have happened okay. And at a point where it actually is going to make a problem that this fellow wants to do it in AB order or some other processor wants to do it in A order it is going to stop there transaction at that moment.

So it is always looking for an actual conflict to happen rather than assumption that conflict might happen somebody might do it in other order this is at random typically needs some hardware support. So if the hardware does not have any support for it then so on software basis you can hardly do this. There are other weaker models that we will look at least few of them where you say am not going to worry about this global strict not the strict consistency but global sequential consistency where exact order for every transaction has to be maintained okay.

**(Refer Slide Time: 14:13)**



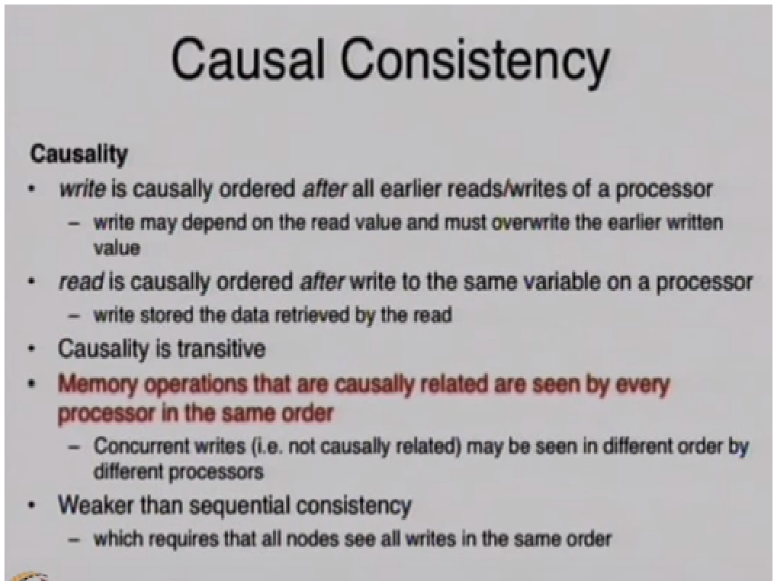
## SC not always Required

```
initially:    ready=0, data=0

  thread1          thread 2

data1 = 1        lock(L)
data2 = 1         sav1 = data1
unlock(L)        sav2 = data2
```

And there are situations in which you would not care especially if you are maintaining some order in a semantic way. Here is an example where you got thread 1 and thread 2 they have a lock thread 1 says here is something in data 1 here is something in data 2 and now I am going to unlock something so that somebody else may lock this region critical region.

I lock set the values unlock only then can somebody get that lock fetch those values that I wrote and then do something in this case those can be re order right data one data 2 can written in any order okay. We can be read in any order because semantically I have ensured that does not matter so there are these kinds of generalization that are maintain one of them are simplification it is rather that are and one of them is re-consistency model that we have seen before in the context of open MP.

**(Refer Slide Time: 15:31)**



## Causal Consistency

**Causality**

- *write* is causally ordered *after* all earlier reads/writes of a processor
  - write may depend on the read value and must overwrite the earlier written value
- *read* is causally ordered *after* write to the same variable on a processor
  - write stored the data retrieved by the read
- Causality is transitive
- Memory operations that are causally related are seen by every processor in the same order
  - Concurrent writes (i.e. not causally related) may be seen in different order by different processors
- Weaker than sequential consistency
  - which requires that all nodes see all writes in the same order

But there are some in between weak and sequential you might this is saying that if you keep the lock their you can reorder those data and nobody will care. So here is an example because something were even if you have violate sequential consistency the program will be it is not sequential consistency may be a over kill motion type you do not have always need a sequential consistency.

And as a matter of fact that is why essentially no multi-processor really guarantee sequential consistency. So there is this notion of causal consistency which says that let us relate operations that is depend on each other let us not say every operation and every access that make has to be ordered in the order specified them I made them in you only do this for things that depend on each other.

So there is a notion of causality you say if I write something then this write will has to be ordered after all the other access so write is somehow discussion why after all accesses. So we are still going to leave linearize the reads and writes. In global sequential consistency said that must respect my order now I am saying that need not respect of my order but these are two orders must respect.

If I have a write that write must be ordered after everything must came before it okay everything because whatever I had read is in some state that determine this write this written value may be function of everything I have read. So anything conflicting or non-conflicting anything I have read before or written because if it write in wrong order people will form order. Anything has happened before our right has to happen before in the final order in the final sequential order.

But the read is only ordered with respect to its consecutive it is conflicting right if I read some variable X and you have written the variable X before or other I have written that variable X before then that written has to come before write than read no first written only says write has to be ordered before after everything before it not about what after it this can come this can reorder it with respect to it except if it can flex.

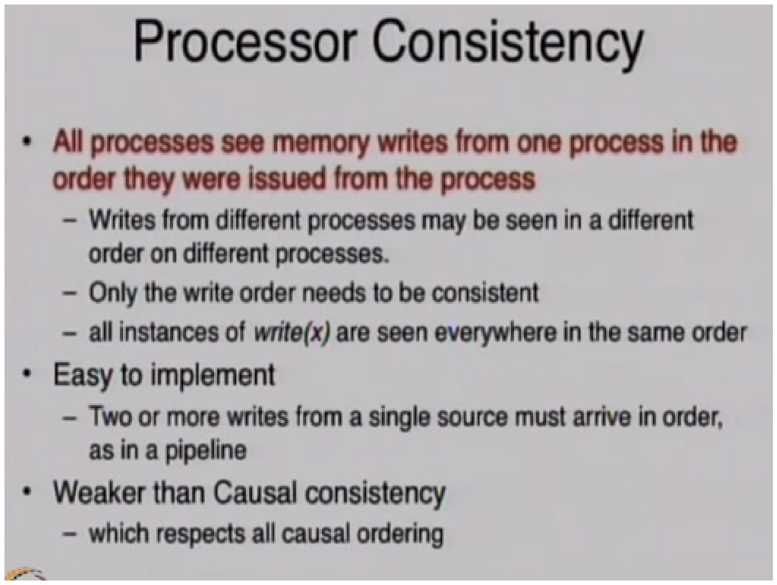
So write of something so write of X read of Y can have read go there then I do not care okay. And then of course a little transitive if I have to come before you and you have to come before somebody else than I better come before somebody else. And only these two kinds of relationships of every single program needs to be maintained okay. So eventually our goal is to t create and order global order which respects every processors order right in this case it does not have to respect all the orders of every processor only the causal orders.

It must respect the causal orders of processor 0 causal orders of processor 1 and so on so that inter-processors it is same thing we did in sequential consistency right. If my value was read by that person then the final order must put that read after my write. So final order must be equivalent to some sequential order right different sequential order will get different results of the program but their all consistence right.

What result you want is up to the program if you some specific reorder you got to synchronize but as far as the memory contract is concerned it says that as long as this order can be given so that everybody sees the consistence order memory is done if you want some more semantics on top of it that program is responsible. How can read and write conflict if you are doing the same variable so in the variable that is the definition.

So if this processor reads X and write X right not consecutive necessarily with respect to every other so for a right it says that all the access before it have to be kept before and for a read it says that all the accesses before it which are rights on to the same variable have to be kept.

**(Refer Slide Time: 21:46)**



## Processor Consistency

- All processes see memory writes from one process in the order they were issued from the process
  - Writes from different processes may be seen in a different order on different processes.
  - Only the write order needs to be consistent
  - all instances of *write(x)* are seen everywhere in the same order
- Easy to implement
  - Two or more writes from a single source must arrive in order, as in a pipeline
- Weaker than Causal consistency
  - which respects all causal ordering

Bit weaker version which is called as processor consistency which is something that actually many hardware support including intel for a des not mean that they provide it something that can be supported when you put it on multiple machines and the processor consistency essentially you do not relate all kinds of causable access it is simply says that the memory access that are write from a given processor must be put in the same order okay.

So if processor A writes A, B, C, D then everybody must see A happened before B before C before D okay. There is so just that by itself what is called consistency processor consistency little bit stronger than that where the writes for the given variable X must be seen in the same order everywhere if this fellow writes X or let say pic variable X all the writes to variable X must be order FIFO consistency everywhere right.

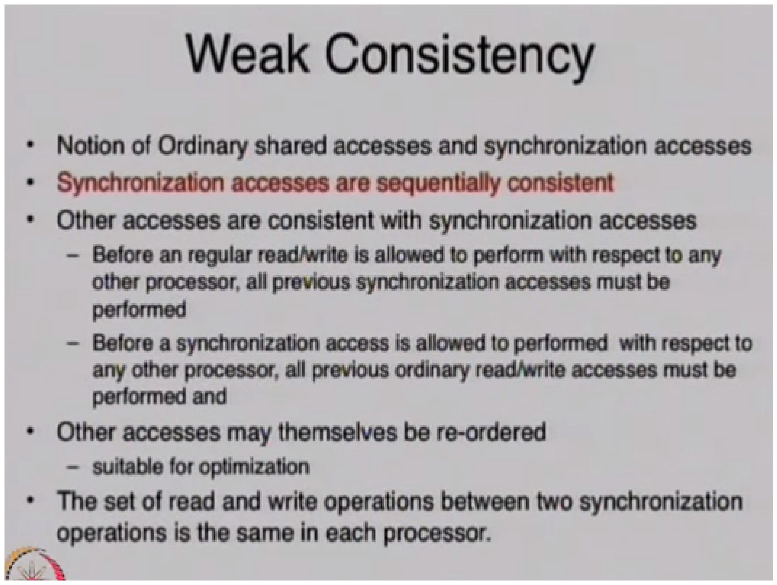
Consistently meaning ordered with respect to process A rights processors B writes processors C writes processors B writes okay. If you remove that when you simply say all the writes happen in auto issued there not become what is known as FIFO consistency it is coming back you remove that restrictions and FIFO consistency is relatively easy to remove where you remove even the second restriction basically writes or pipeline.

In fact that is why is also sometime as pipeline consistency or FIFO consistency and the FIFO also suggest basically the something. When I first write a processor mix will get effected or will complete before a second write the processor mix so the processor writes along the way cannot cross each other that is all okay. So relationship with reads as gone away as a result of which this is weaker and processor in fact as not much more to do says all the writes can pipeline writes cannot overtake each other reads can come back from cache early and it is still fine okay.

It is now become the more lakhs the consistency model becomes harder it becomes the program programmer will sure that your semantic is maintained right. That means you have to make sure the everything is synchronized okay. Both meaning processor and FIFO am saying that writes to be ordered so even if it is intercourse if I am saying my that write should be ordered that means for piece of one it is.

The write it is issued from the same processor or you take the sub set of writes issued by processor all the accesses come in the form of this is the write to variable heads that processor Y right. So write have to kinds of subscripts associated with it write so once is with the  processor the other is with the variable only the processor is that as to be consistent.

**(Refer Slide Time: 26:42)**



## Weak Consistency

- Notion of Ordinary shared accesses and synchronization accesses
- Synchronization accesses are sequentially consistent
- Other accesses are consistent with synchronization accesses
  - Before an regular read/write is allowed to perform with respect to any other processor, all previous synchronization accesses must be performed
  - Before a synchronization access is allowed to performed with respect to any other processor, all previous ordinary read/write accesses must be performed and
- Other accesses may themselves be re-ordered
  - suitable for optimization
- The set of read and write operations between two synchronization operations is the same in each processor.

And finally the weak consistency model where there is an explicit there is a special type of access that is called as synchronization access right and says that if you want your access to be synchronized to be consistent then you make the special type of access you do not simply say read X you say read consistent X or something like that is it and the read consistent and the write consistent will be synchronize with respect to each other will be sequentially consistent with respect to each other all the other reads they can happen which ever order okay.

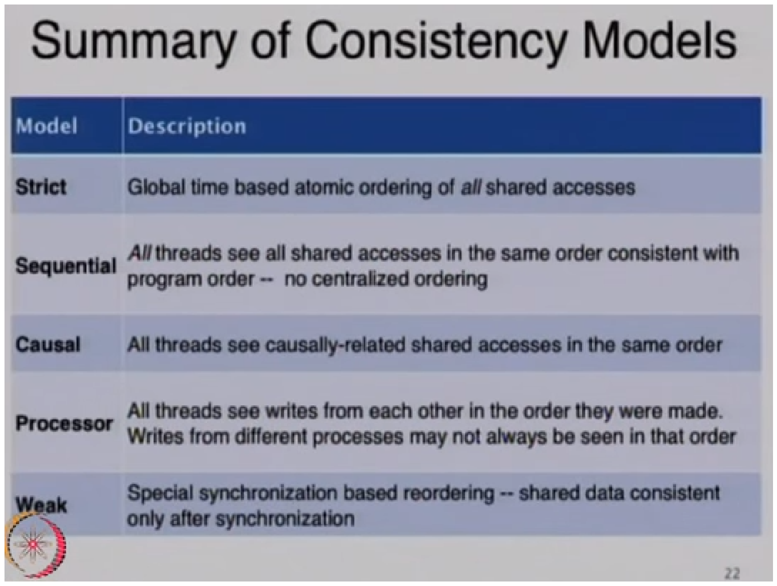
And that is in fact what open MP does right so it is like normal read and write or not memory access at all. When you make the special consistent read and consistent write access only those need to be serialized others can be seen by different processors and different order does not matter. There is some semantics the programmer wants this read happens before that read and then it is programmers responsibility to either call it synchronize or call it synchronized read or make a law.

You do the read synchronize whatever barrier whatever you synchronization method may be then these other read. So the special type of synchronization access for open MP is a flushed something. When you say flushed something that means the read and writes to that variable have to synchronized and of course there are some thing that are implicitly flushed barrier variables flushes that happen at the end of parallel construct for example but either implicit or explicit each access is either a flush access or non-flush access.

Meaning its either a synchronization access or a non-synchronization relations access then only the flush accesses will get consistent with respect to each other it is even weaker than read consistency because not flush of A here and flush of B here will be made consistent. These two flushes will be made consistent this as flush AB and this is flush AB. If these two flushes have an intersection then they will be made consistent with respect to each other otherwise it is like non-synchronized access okay.

And so this is within a processor it has to be consistent right within the processor you read something and then you do a flush read then they will that processor has to be see this happen before that happen no within a processor it is like sequential environment right. So automatically anything that processor sees has to be in that order but among the processer where you serializing all the accesses these processors the non-flush or synchronize accesses not exist. They do not have to be put in a specific place among the list of flush okay for that processors.

**(Refer Slide Time: 31:26)**



## Summary of Consistency Models

| Model | Description |
|---|---|
| Strict | Global time based atomic ordering of *all* shared accesses |
| Sequential | *All* threads see all shared accesses in the same order consistent with program order -- no centralized ordering |
| Causal | All threads see causally-related shared accesses in the same order |
| Processor | All threads see writes from each other in the order they were made. Writes from different processes may not always be seen in that order |
| Weak | Special synchronization based reordering -- shared data consistent only after synchronization |

22

So here is the quick summary there is strict which is essentially it is possible because you say that it becomes instantaneous you make the read call and that time you made the read call as well it happens so everything is deterministic. In sequential consistency it is equivalent to some sequential at setup operations and if you want your sequential operations it is his business.

Equivalent to some sequential set of operations but it has to be consistent for all operations of each processor okay. And causal says not all operations have each processor only related process of each processors this special causal relationship it has to be respected and then that related definition is weaken further by saying that in processor consistency specific writes and reads are related and in weak consistency only the flushed are synchronized reads and writes are related okay.

The implementation of inter architecture does consistency okay but you have to remember that is an almost irrelevant question where you are programmer. Because you have not directly calling processor instructions right there are fence instruction that it uses to make sure that these are consistence that their accesses are consistent but between the hardware and you sitting at open MP their if you using it a compiler layer so they are all this layers together provide you some guarantee of one or the other consistency.

So for example if you using open MP right it says am going to guarantee to you this special type of weak consistency right so it is the entire environment which provides that guarantee that contract. So whatever the end of line hardware supports is somewhat irrelevant but Intel hardware does support if you want to build an new open MP which provides processor consistency you can do that.

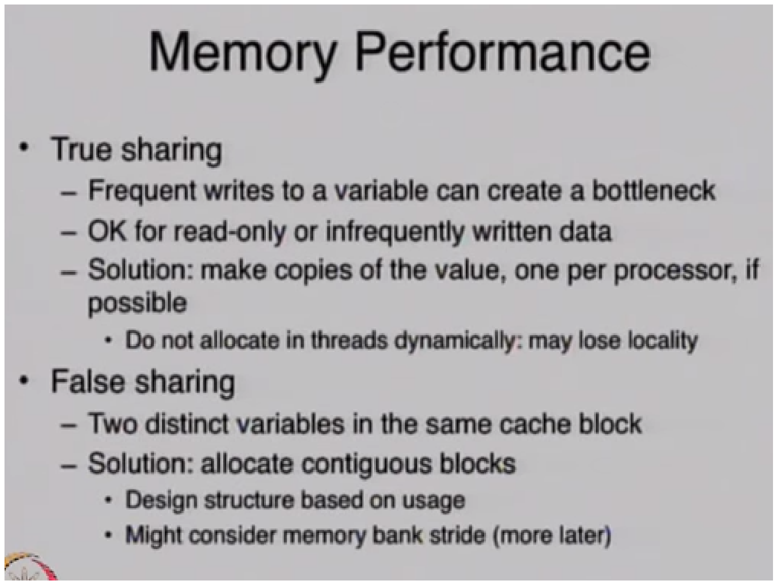
There is no results on their relationship on their timing right but one is clearly stronger than the other right where you are saying that thing being relaxed in this so this is strictly more strict than this those are to be ordered like you said this additional ordering has to be maintained because I want some more constructs.

So there is a strict precedence that is harder things to say right because weaker model that is not all of them that is part of it meaning that there may be features like locking test and set instructions other things that you can loose to encapsulate that accesses to provide a stronger consistency model for example if you wanted sequential consistency it can be weaker one is automatically implemented when you implemented the stronger one but to implement the stronger one you have to pay some price.

It can be done but you may have to lock they have to synchronize they have to do some special exactly not equivalent they are just different environment right if with the weak you are saying some other features they together can implement something strong alright. So now we are going to get we talked about the consistency and to some extent we are talked about the fact that one is faster to implement then the other will talk about general programming performance issues shared memory or memory that is distributed across multiple processors.

And then get on to some of the issues in designing parallel programs okay so one of the things that if you are aware of and if you account for in your program you can make a significant difference to its performance it is called false share okay.

**(Refer Slide Time: 37:31)**



Memory Performance

- True sharing
  - Frequent writes to a variable can create a bottleneck
  - OK for read-only or infrequently written data
  - Solution: make copies of the value, one per processor, if possible
    - Do not allocate in threads dynamically: may lose locality
- False sharing
  - Two distinct variables in the same cache block
  - Solution: allocate contiguous blocks
    - Design structure based on usage
    - Might consider memory bank stride (more later)

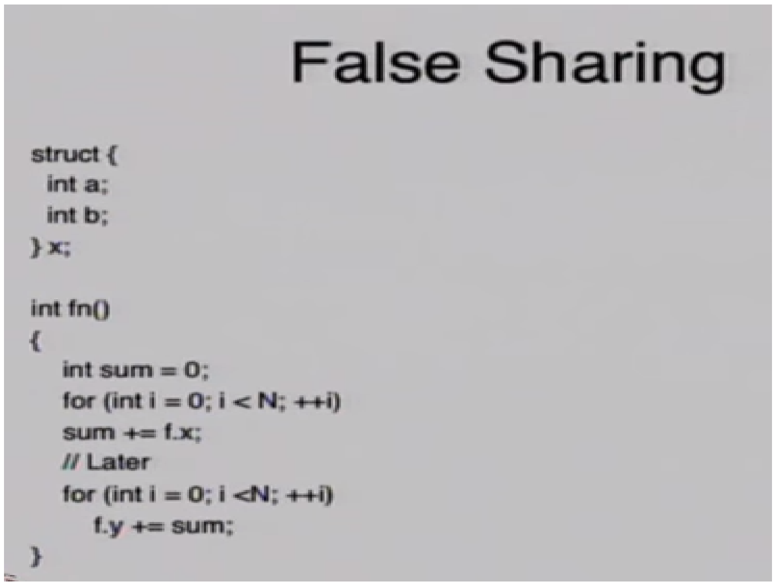
Meaning that everything is set so far read access reflecting with this read X in reality gets mapped to read of a cache line where belong is conflicting with read of a cache line where Y belongs to right because entire cache lines are flushed at a time. So if have one element that you have written into a cache line that entire cache line is going to get flushed okay or if you reading one element from cache line that is not in cache one element that is not in cache then that entire cache line will be fetched.

So this is called false sharing I said I have variable X and variable Y and I have written variable X and I have read variable Y and I assume there is not conflict okay. But internally there will be as a result of which it has to be maintained sequential or whatever the consistency is providing but the fact that it has to write something into the main memory when you only want to write one part of it means there is more synchronizing happening you thought right or you expect it.

So when that happens when you have two variables in the same cache lines it is called false sharing versus same variable I am reading some variable here reading that variable or writing that variable again later. There are I can give you some type solutions that are listed on the slide but there is not really good way to detect or eliminate or systematic I should say to make sure that you do not or reduce in any sense false sharing okay.

But mostly what you can do is allocate things in certain this will become very clear and important rather you start talking of CUDA where we are very aware of the memory or key and the memory structure let me get you an example.

**(Refer Slide Time: 40:12)**



## False Sharing

```
struct {
  int a;
  int b;
} x;

int fn()
{
    int sum = 0;
    for (int i = 0; i < N; ++i)
    sum += f.x;
    // Later
    for (int i = 0; i <N; ++i)
        f.y += sum;
}
```

Here is a strut which has variable A field A and field B okay I have got an array of this strut and somewhere I say goes sum up all the values all the A values in the strut so far I = 0 to N sum + = f.x and then little while later I go do something to all the while fields okay am adding this case sum to each Y member I let me fix this when you going through the first loop you were jumping all the B's.

A and B are continuous the entire structure of the memory is A, B okay so this is what is also known as an array struts and alternative might have been called a strut of array structure of arrays where you say I have get an array which is A and an array which is B and a strut which is this two members an array of A and array of B okay.

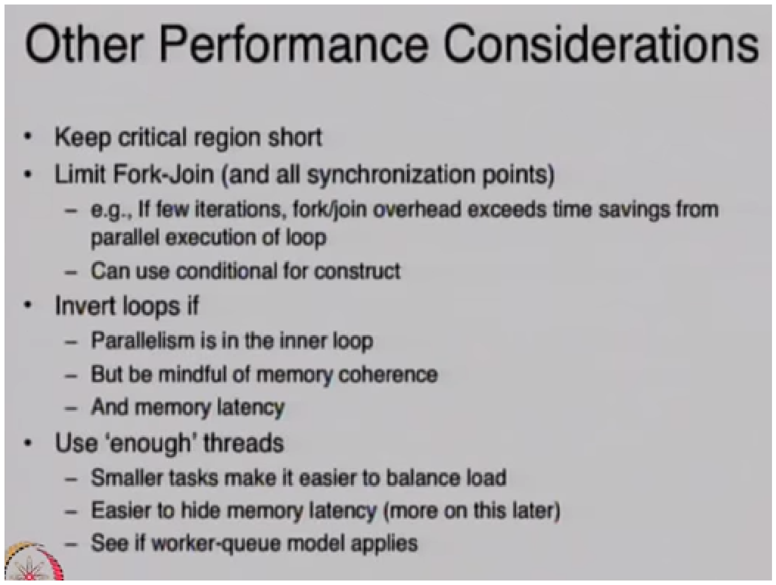
So in this case you are paying the price for accessing B in the first loop and you are paying the price for accessing A in the second loop because in all likelihood the A and B belong to the same cache lines are whenever A is being fetched B is being fetched okay. In this case if you had this is what you have to do you could eliminate false sharing by using in this case again a strut of array.

All the A are together and all the B come later so now if you are reading variable is in A the members in the array A then you getting good utilization of the cache line read some of few members of A processed in few members of A again process them and so on that battle is always there and parallel programming only makes it even worse. And the truth is that you can imagine some of it actually done also a compiler figuring this array and this reordering in B.

It is actually very hard research goes on in how to do that consistently without introducing error in all context cannot be done as of today I do not know any of that does that would be a significant change reordering access for example if you have a loop two dimensional array that goes for I = 02 and J = 02M that can be reordered in some context that can be is possible and done.

So outer loop is going through rows inner loop is going through column you translate them and how loop goes through column inner loop goes through rows in order to make the cache behavior better meaning false sharing reduced okay.

**(Refer Slide Time: 44:24)**



## Other Performance Considerations

- Keep critical region short
- Limit Fork-Join (and all synchronization points)
  - e.g., If few iterations, fork/join overhead exceeds time savings from parallel execution of loop
  - Can use conditional for construct
- Invert loops if
  - Parallelism is in the inner loop
  - But be mindful of memory coherence
  - And memory latency
- Use 'enough' threads
  - Smaller tasks make it easier to balance load
  - Easier to hide memory latency (more on this later)
  - See if worker-queue model applies

Some other high level performance considerations clearly you have to keep critical regions shot if specially if you have lots of them and meaning that when somebody is in critical region it is probably stopping somebody else from coming there and that somebody else is sitting waiting to that method okay. So you should keep them as short as you can these are just other performance consideration that is why is tighten.

And in fact will go through a few column text are whatever through this discussion of parallel programming that are more from a experience then from anything else. Again all kinds of synchronization so the critical region is an one example essentially want to limit your synchronization has much has possible will talk about it in more detail soon enough but you can essentially batch synchronization you can say instead of saying do this make sure everybody has done that you can do this make sure everybody has done that.

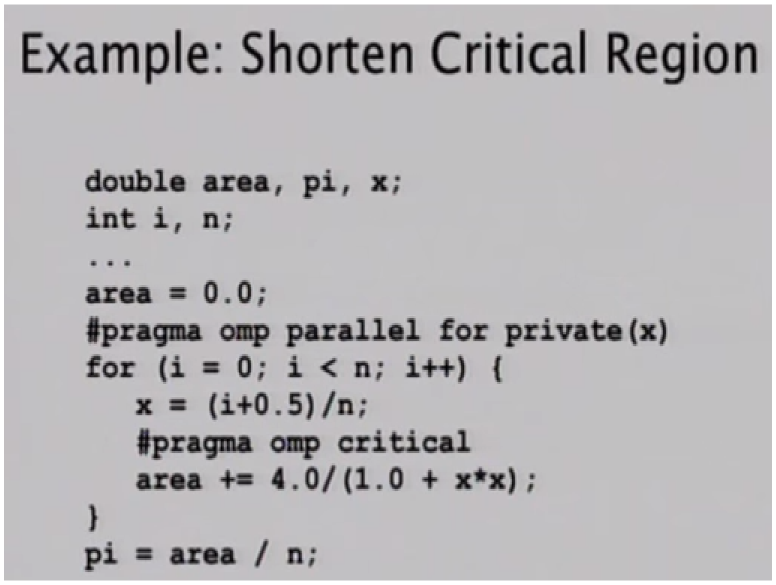
You can say let us do this much make sure everybody has done that much right so you are basically reducing these synchronization per computation overall okay. So that is ultimately the goal you cannot eliminate synchronization but for each synchronization the best use of it take the most advantage out of that synchronization. We are just talking about loops we can say for example in pair for it is possible that outside loop as communication but inside loop is parallel there is no every iteration of the loop working on independent unit of data.

You can reorder those loops so that those outside loops have independent units of work and each work is now lots of dependent things can be done sequentially on each processors okay. And then finally use enough threads nobody knows what that number is it depends on the architecture it depends on your program it depends on the overhead and communication versus the computation type but again I think I have mentioned that it you essentially use rule of the thumb that we have at least 4 to 5 times active threads at any given time.

So that you can hide latency if there are pure compute jobs you are doing nothing but taking too values in the register and doing some operation on them which is quite rare then you do not need to many threads but as soon as you start to do communication mixed to computation memory accesses mixed with computation you want to have may be 4, 5 may be ten times the number of threads has have number of processors okay.

And sometimes so I said active threads 4 to 5 times the active threads sometimes when you breaks your task down you may actually have logically many more say I have got work here divided among thousand threads having these ten threads will do ten of those ten of the thousand work and anybody works done and some more okay that is another way of organizing your computation talk little bit more of that.

**(Refer Slide Time: 48:45)**



## Example: Shorten Critical Region

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
   x = (i+0.5)/n;
   #pragma omp critical
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Here is one example of how you might want to shorten the critical region what does this do in the critical region you are adding something into the shared variable which is hopefully being or as presumably being added to or operate upon by somebody else who also wants to be in the same critical region who is protected by a critical. So you say in this critical region area is going to get added with four over 1 + X square right.

Seems like a small enough critical region but not the smallest you only need access to area inside the critical region why are doing all the other computation move all that outside and only assign the variable. And of course there are many other features like atomic operations you might use that still it eventually it amounts to having the critical region atomic operations are equivalent to critical regions but the its really narrow then when the down is going to happen.

So that critical region is going to have further bound in terms of the range of clocks where that is after okay in fact I did reduction here I was meaning to do atomic where the atomically add do the area. Reduction is an another example where you simply eliminate the critical region but then there is baggage as you pointed it may not be being done in parallel.

You have note that in this example it is anyway not being done because everybody was waiting to add is value and so the reduction is not being done in parallel at least you got read of everybody waiting for that long period. Overhead is avoidable but generally cannot avoid not always right that is this read off is to make between how often we have to synchronize versus if you have to wait for that long to synchronize is somebody is waiting for much earlier same synchronization and hence ideal for long period.

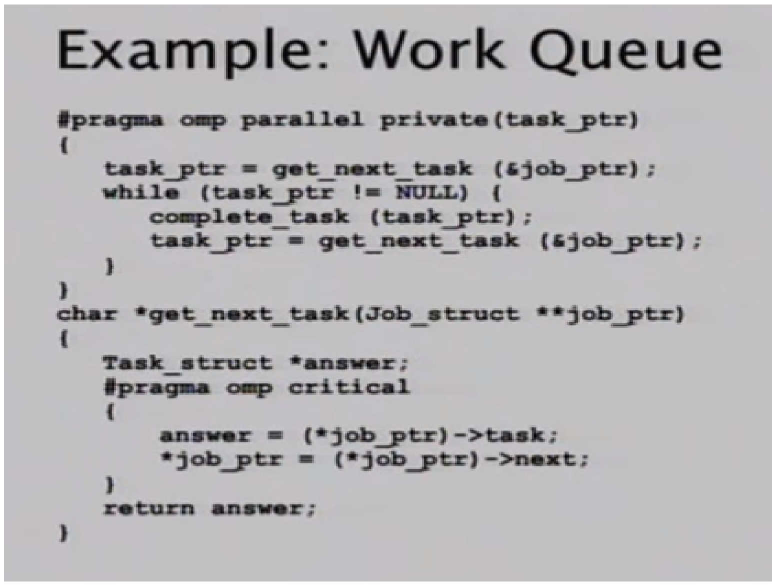
It depends right if you can club to synchronization things it can help sometimes but you do not club doing lot of unrelated work in between typically club is work all of been related to synchronization in that if you can say am taking this 10 synchronization step returning to one where all ten where needed that is not the case here right that can help I that is the discussion it just having sometimes it want to say instead of do this synchronize do all of these and then synchronize right that does make sense.

But you still have to enough useful words between synchronization and then you cannot simply club everything because people are sitting at different speeds and if you synchronize their then the level of idealness cross the system is potentially going to increase okay because they will not they are not contending with each other so if you club them again there is inspection of code needed to make sure that you are making the right decision.

But typically you would expect that you do not want to club unrelated critical sections because you are basically adding synchronization things it is not going to wait for this region are not going to wait for it right. Again this work model again will come back to it again it is more of a how do you manage specially when work gets dynamically generated sometimes you say I have for this thousand things to do I have got this 5 process to do them could do right.

Another time say I have got one thing to do but while doing that I will figure out what more is to be done right. And so work is getting generated dynamically in specially in case is like that work is become very important or useful rather.

**(Refer Slide Time: 54:45)**

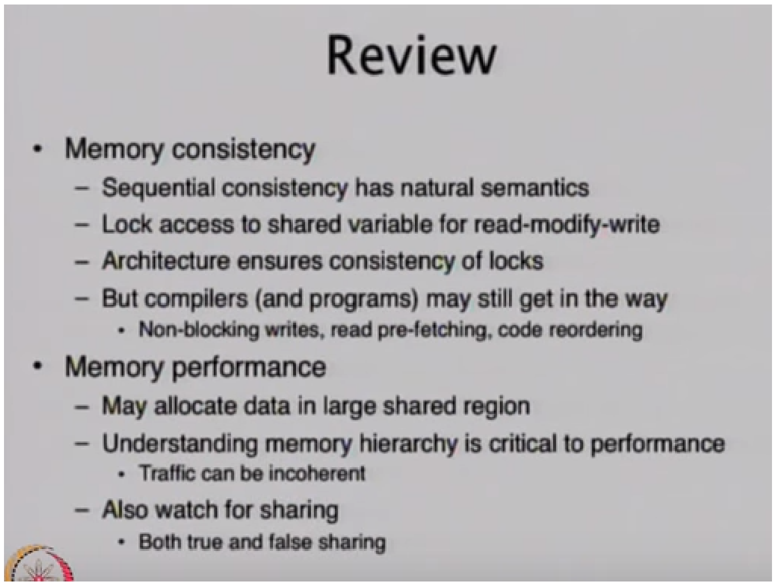


## Example: Work Queue

```
#pragma omp parallel private(task_ptr)
{
   task_ptr = get_next_task (&job_ptr);
   while (task_ptr != NULL) {
      complete_task (task_ptr);
      task_ptr = get_next_task (&job_ptr);
   }
}
char *get_next_task(Job_struct **job_ptr)
{
   Task_struct *answer;
   #pragma omp critical
   {
      answer = (*job_ptr)->task;
      *job_ptr = (*job_ptr)->next;
   }
   return answer;
}
```

Where you simply have some kind of loop where you say if there is some more work let me take the next things that available to do of that is ready to be done and let me do that okay and all the processors are doing the same thing of course there is issues with synchronization right.

You do not want people to take same work you do not want some work taken by anybody so those things will automatically done through is a critical region will look later the entire queue as to be critical region has reduced that critical region to a small instruction section or probably eliminate it altogether there is more details on the same thing I will probably skip through this code is not really adding much.



## Review

- Memory consistency
  - Sequential consistency has natural semantics
  - Lock access to shared variable for read-modify-write
  - Architecture ensures consistency of locks
  - But compilers (and programs) may still get in the way
    - Non-blocking writes, read pre-fetching, code reordering
- Memory performance
  - May allocate data in large shared region
  - Understanding memory hierarchy is critical to performance
    - Traffic can be incoherent
  - Also watch for sharing
    - Both true and false sharing

Especially we are not going to come back to this notion of work use and so review in the context only of memory so you have been talking about memory consistency memory performance we talked about little bit about lot balancing or keeping people non-ideal working will focus on that side how do you keep how do you distribute task how do you distribute assign he task how do you make processors in next task can all that.

But in the memory side you have got to understand the consistency model right and paste on that model you have figure out what is the semantics you want and synchronize access where there is that part of the programmers choice once you have done that it may look at where you can eliminate slowly by reducing false sharing reducing critical region and like any questions on that and then I typically people who talk about computation parallelization then if at all talk about memory okay.

I have done the opposite on purpose because memory optimization is very often be ignored and is an equal partner if not in some context the more important partner in getting the maximum speed get many of the anymore okay so the next thing we will talk about computation parallelization but we will stop here and we will continue