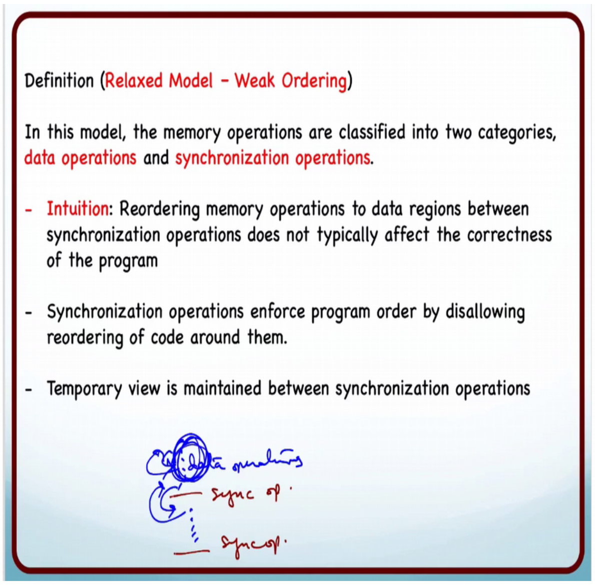
Introduction to Parallel Programming in OpenMP

Dr. Yogish Sabharwal
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi

Lecture – 11
OpenMP Shared Memory Consistency Model

(Refer Slide Time: 00:27)



Definition (Relaxed Model – Weak Ordering)

In this model, the memory operations are classified into two categories, data operations and synchronization operations.

- Intuition: Reordering memory operations to data regions between synchronization operations does not typically affect the correctness of the program
- Synchronization operations enforce program order by disallowing reordering of code around them.
- Temporary view is maintained between synchronization operations

So, one relaxation to the sequential consistency model is called the weak ordering. It is a relaxed model. So, what does this model say? It says that the memory operations are classified into 2 categories; data operations and synchronization operations, the intuition is that the one segment of data operations the operations that are happening in one data operation before some synchronization operation do not affect the data operations that are happening after the synchronization operation what does it allow the compiler to do.

So, the reordering memory operation to data regions between synchronization operations does not typically affect the correctness of the program. So, what does it mean that within the segment of data operations, I can move instructions around without impacting the correctness of the program, all right.

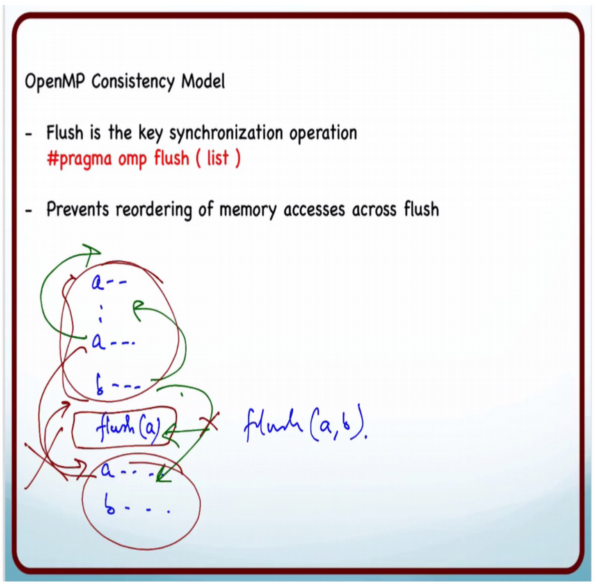
So, now what can the compiler do? Now within a data segment. So, you will have some data operations. So, these are all data operations and then you will have some sync operation, then again, you will have some data operations and then maybe again you have some sync operations and now the compiler knows that it sees the sync operation it is aware of these synchronization operations and it sees these data operations and it knows that it can move these instructions around as it wishes it is not being to impact the correctness of the program that is the agreement between the programmer and the system that we did the data operations, I am allowed to reorder as I wish as long as I do not see a sync operation I should not reorder across the sync operations, right.

So, I should not move any instruction from here to here or here to here it should not cross any boundary, but between these instructions, I can reorder right and I can keep the values in the registers as long as I do not hit the sync operation right if I when I hit the sync operation then I need to be careful about a lot of thing whatever I have in my registers, I need to write them all back to the memory, right.

Any updates which are pending in any write buffers or something need to complete, right. So, a lot of things have to be taken care of when the sync operation happens because at the point of the sync operation the state of the processor has to be consistent with the memory.

So, synchronization operations enforce program order by disallowing reordering of code around them right and essentially a temporary view is maintained between synchronization operations. So, it is between the synchronization operations there may be a view that the that particular processor has which is not the same view as any other processor of that memory right I may have a local in the register which the other processes is not able to see. So, it is a temporary view.

(Refer Slide Time: 03:12)


OpenMP Consistency Model

- Flush is the key synchronization operation
  #pragma omp flush (list)

- Prevents reordering of memory accesses across flush

a - -
:
a - - .
b - - -
flush (a)
a - - .
b - . .
flush (a, b).

What is the openMP consistency model? Well, it is the weak ordering with further relaxation.

So, we will talk about that, but first let us understand that with respect to weak ordering there is a key synchronization operation this is called flush. What it prevents is? It prevents reordering of memory accesses across the flush instruction; all right, you can give a list of variables to the flush instruction. So, what does that mean; that means, that suppose I am doing something with a reading writing something I do something with a, I do something with b, right and then I say flush a and then I do something with a I do something with b.

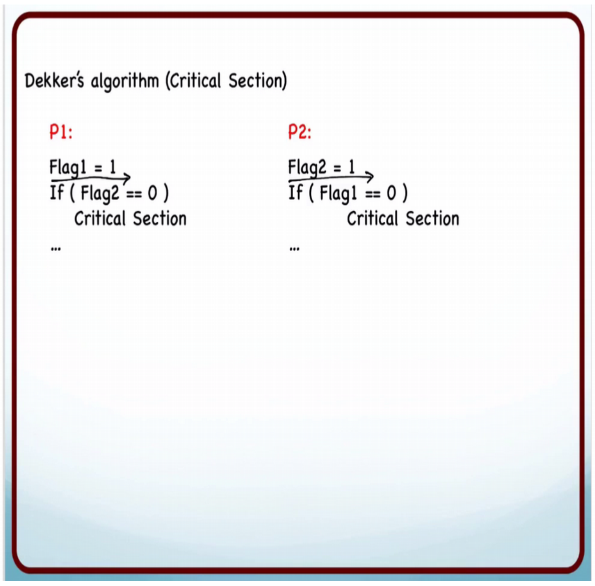
So, this flush ensures that I cannot reorder the instructions which access a across the flush instruction I cannot move this instruction here I cannot move this instruction here these are not allowed what is allowed. So, moving this instruction before this instruction is allowed I can do that I can move this b before this a axis, I am allowed to do that.

As a matter of fact, I can even move this be down here why because this flush only wants a to be synchronized, right. So, it does not care about b, if I say flush a comma b, then even this will not be allowed, right, even b cannot be moved across this if I say flush a comma b, it will ensure that all these instructions get executed then flush gets executed, then all these instructions get executed.

Student: only if a is not depending on b then only the statement can be moved like this.

Yeah, I mean there are lots of other things that you have to consider before moving any instruction around right that the compiler figures out. So, we are not getting into that discussion that what all the compiler is allowed to do the compiler will take care of lot of things before moving any instruction around, it has to take care of a lot of things dependencies, right and a whole lot of stuff. So, we are not getting into that discussion we just think that if the compiler wants to move this around is he allowed to move it around or not and this flush instruction says no you are not allowed to move it around clear.

(Refer Slide Time: 05:28)



Dekker's algorithm (Critical Section)

```
P1:                      P2:

Flag1 = 1                Flag2 = 1
If ( Flag2 == 0 )        If ( Flag1 == 0 )
    Critical Section         Critical Section
...                      ...
```

Let us try to fix this code Dekker’s algorithm for critical sections or this is similar to the code we saw except for now; it is actually what it is meant for critical sections, right. So, where should I put flush instructions in order to ensure that this code works properly and what is the flush instruction that it should use.

Suppose that we are no longer in the sequential consistency model right now we are in the week ordering model where now the compilers allowed to move things around now I am asking the question in openMP where do I have to introduce the flush instructions. So, that the compiler does not mess this up otherwise, it is going to move this flag one down and then both of them are going to enter the critical section together.

Student: (Refer Time: 16:19).

We do not want that to happen.

Student: Between the critical section and the; if statement.

Between critical section and the If statement, here, this does not help, right, if both of them enter the If condition the damage has already been done.

Student: Between It and assignment.

Between if and the assignment, yeah. So, I need to add a flush here and I need to add a flush here; what is the flush instruction I need to add?

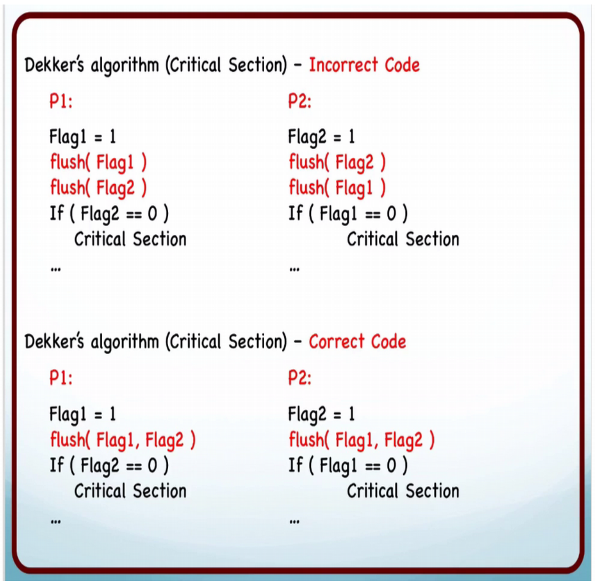
Student: (Refer Time: 06:50) flush flag (Refer Time: 06:51)

Flush flag have to that way the easiest way out, right, you do not know which one to do. So, they will do allow them.

Student: (Refer Time: 07:00)

That is the safe way around.

(Refer Slide Time: 07:02)



Dekker's algorithm (Critical Section) – Incorrect Code

```
P1:                          P2:

Flag1 = 1                    Flag2 = 1
flush( Flag1 )               flush( Flag2 )
flush( Flag2 )               flush( Flag1 )
If ( Flag2 == 0 )            If ( Flag1 == 0 )
    Critical Section             Critical Section
...                          ...
```

Dekker's algorithm (Critical Section) – Correct Code

```
P1:                          P2:

Flag1 = 1                    Flag2 = 1
flush( Flag1, Flag2 )        flush( Flag1, Flag2 )
If ( Flag2 == 0 )            If ( Flag1 == 0 )
    Critical Section             Critical Section
...                          ...
```

So, what about this is this code good; well, I have already written incorrect code so; obviously, it is not good what is wrong with this code. So, remember flush how does flush work flush ensures that operations involving that variable or that list of variables is not moved around the flush, right. So, what can go wrong?

Student: I will not reach any of the critical section.

It will not reach; why?

Student: (Refer Time: 07:40).

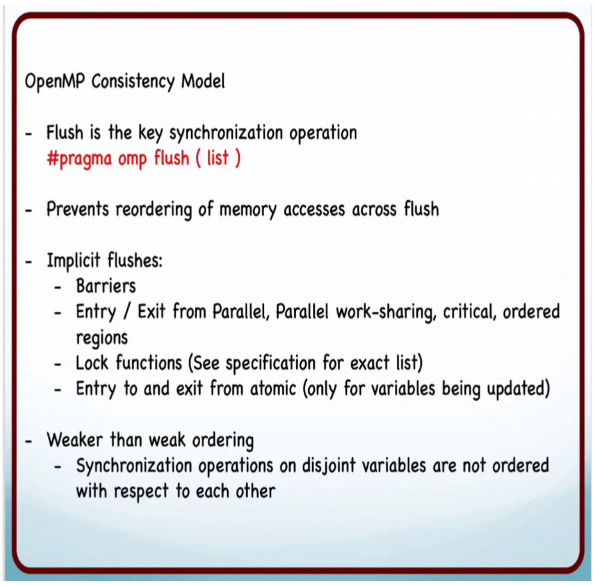
So, you understand this is not a barrier right you understand what a barrier is barrier is where both processes must come together before proceeding further this is not a barrier a flush is just saying make the memory consistent with what I have that is all; nothing else it does not care where the other processor is.

So, what the compiler may do is that it may pick up these 2 instructions and move them down that is perfectly valid, right, why it has not moved memory accesses involving flag one beyond the flush for flag 1, it has not moved instructions involving flag 2 beyond the flush for flag 2, but its move them both together that is allowed the compiler is allowed to do that right.

So, you see the problem and now both of them can enter the critical section by the same sequence of code instructions that we saw earlier. So, what is the correct code flush flag 1 flag, right that is the correct code now it cannot move both of them down because then it will not maintain the order between the flush of flag 2 and the if condition of flag 2, right.

So, you need to go back and just look at these examples and study them very very carefully, right, the more you look at it the more you will understand why it is the way it is.

(Refer Slide Time: 09:20)


OpenMP Consistency Model

- Flush is the key synchronization operation
  #pragma omp flush (list)
- Prevents reordering of memory accesses across flush
- Implicit flushes:
  - Barriers
  - Entry / Exit from Parallel, Parallel work-sharing, critical, ordered regions
  - Lock functions (See specification for exact list)
  - Entry to and exit from atomic (only for variables being updated)
- Weaker than weak ordering
  - Synchronization operations on disjoint variables are not ordered with respect to each other

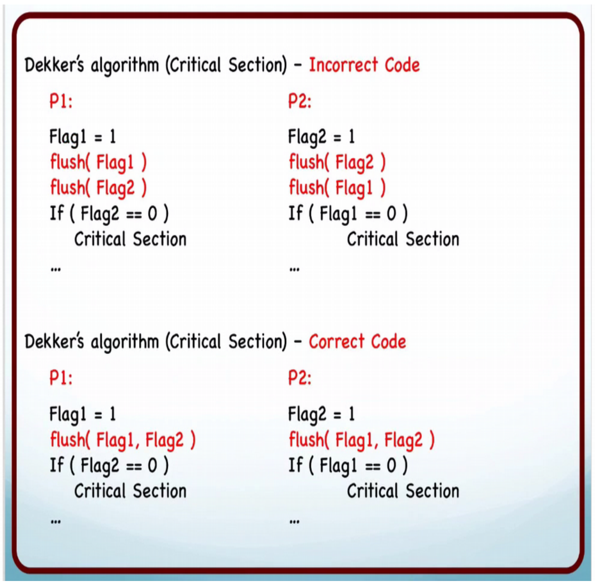
So, we said ppenMP is a weak consistency model and this is further relaxations which we have not come to yet, but the crux is that we have written. So, many openMP programs and we never bothered about flush why is that we never talked about flush and still be wrote program and we run fine and everything seemed to work fine why is that.

Because openMP implicitly puts flush instructions at various places where all does it put the flush at barriers wherever it sees a barrier omp barrier entry exit from all the parallel regions right wherever the parallel work sharing sections are there; hash pragma omp for right hash pragma omp single, it puts flush instructions around that critical sections it puts flush instructions around that.

So, it carefully puts all these flush instructions. So, if you are accessing your data properly then you will not face any issues you will not need explicit use of flush why because when you are accessing a shared variable if you are updating a shared variable you will use a critical section. So, that critical section ensures that the flush is happening, right. So, if you are careful about all your data accesses whenever your accessing shared variables that are using critical sections or atomic right. So, what are the other implicit flushes around locks and entry to and exit from atomic right hash pragma omp atomic remember atomic instructions.

So, whenever you are accessing global variables if you are careful shared variables if you are careful to use hash pragma omp; atomic hash pragma omp critical, right, then you will never land into the trouble where you need to start using flush as a matter of fact using flush is discouraged right you should just always carefully use hash pragma atomic and hash pragma critical and so on, right.

(Refer Slide Time: 11:12)



Dekker's algorithm (Critical Section) – Incorrect Code

```
P1:                          P2:

Flag1 = 1                    Flag2 = 1
flush( Flag1 )               flush( Flag2 )
flush( Flag2 )               flush( Flag1 )
If ( Flag2 == 0 )            If ( Flag1 == 0 )
    Critical Section              Critical Section
…                            …
```

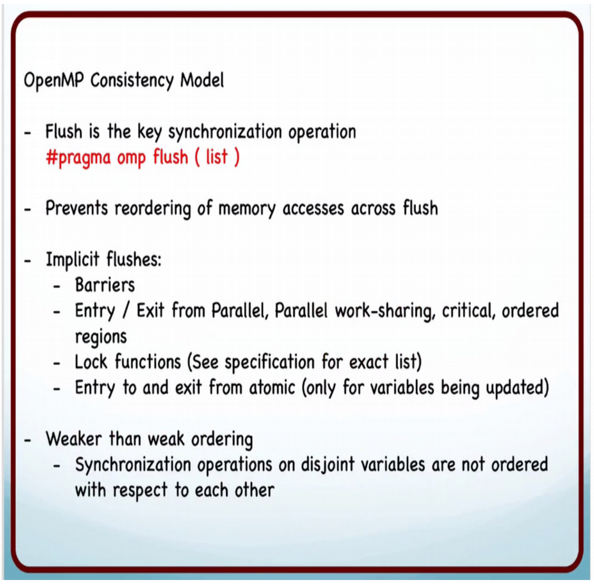
Dekker's algorithm (Critical Section) – Correct Code

```
P1:                          P2:

Flag1 = 1                    Flag2 = 1
flush( Flag1, Flag2 )        flush( Flag1, Flag2 )
If ( Flag2 == 0 )            If ( Flag1 == 0 )
    Critical Section              Critical Section
…                            …
```

So, when will you encounter this issue we saw this Dekker's algorithm, right. So, when do you have to write this flush look I am updating a shared variable flag one and I am not saying critical I mean I am writing code I have not put a critical section around flag one I have not said it as atomic I mean, I am trying to bypass the constructs that openMP has built for me critical and atomic. So, some people do this because they want to write really optimized code, right. So, then they have to use flush, but they have to use it carefully.

(Refer Slide Time: 11:46)



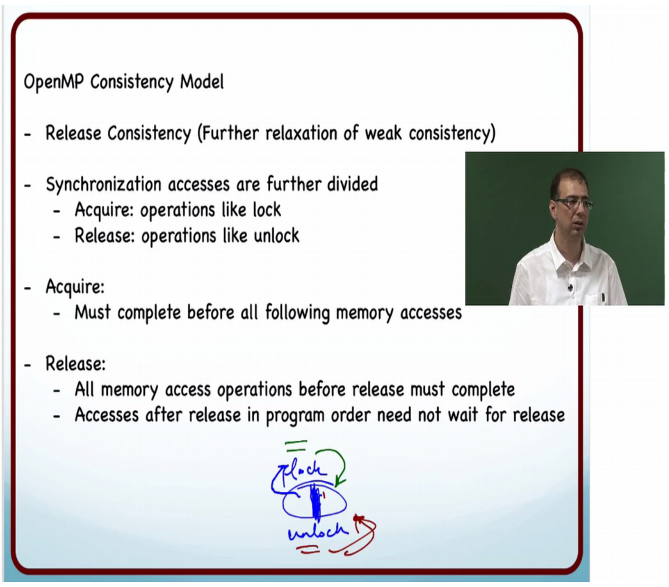
OpenMP Consistency Model

- Flush is the key synchronization operation
  #pragma omp flush (list)
- Prevents reordering of memory accesses across flush
- Implicit flushes:
  - Barriers
  - Entry / Exit from Parallel, Parallel work-sharing, critical, ordered regions
  - Lock functions (See specification for exact list)
  - Entry to and exit from atomic (only for variables being updated)
- Weaker than weak ordering
  - Synchronization operations on disjoint variables are not ordered with respect to each other

So, most of the time, you do not need to worry about flush that is the crux couple of reasons why openMP consistency model is weaker than weak ordering one is that synchronization operation and disjoint variables are not ordered with the respect to each other that we saw right the flush for a did not ensure anything for b memory accesses to b, it did not that say that they cannot cross this boundary right they cannot be reordered around the flush.

That is one way to weaker than weak ordering.

(Refer Slide Time: 12:18)



OpenMP Consistency Model

- Release Consistency (Further relaxation of weak consistency)
- Synchronization accesses are further divided
  - Acquire: operations like lock
  - Release: operations like unlock
- Acquire:
  - Must complete before all following memory accesses
- Release:
  - All memory access operations before release must complete
  - Accesses after release in program order need not wait for release

Another important thing which is more of an advanced concept is that the openMP consistency model the model that it offers is the release consistency model that is a further relaxation of weak consistency. In this what happens is that the synchronization accesses are further divided into 2 kinds of operations acquire and release acquire a kind of operations that happen when you try to get a lock and release operations happen when you try to unlock right and now what this model does is that for acquire it ensures that the acquire must complete before all following memory accesses.

So, what happens when you try to lock and unlock right you are doing something over here, these are the instructions you are trying to protect within the lock. Suppose I have an instruction here and suppose I have an instruction after the unlock will it matter if one of these instructions before the lock completes after the lock has been acquired.

No what is more important is that the instruction inside the lock must not take place before the lock has been acquired that is the critical thing right. So, that is going one level beyond just ensuring that memories can sustained it is saying that when an acquirer operation happens, then the acquire operation must complete before all following memory accesses and similarly when a release operation happens, then all memory access operations that are before the release must complete before the release happens, right.

All these instructions which are between the lock and unlock between the lock and unlock, they must complete before the release happens before the lock is released and accesses after releasing program order need not wait for release, right. So, some of these instructions some of these red instructions may even get executed before the lock has been released does not matter, right, these blue instructions between the lock and unlock are the once that we really care about.

So, this allows even more optimizations to be done by the compiler by the ten time system and so on. So, it is here, it relaxes the constraints even further you understand this is important right because of the compiler and the runtime system do not do these optimizations your code is going run very very slowly.