

**High Performance Computing**  
**Prof. Matthew Jacob**  
**Department of Computer Science & Automation**  
**Indian Institute of Science, Bangalore**

**Module No. # 09**

**Lecture No. # 39**

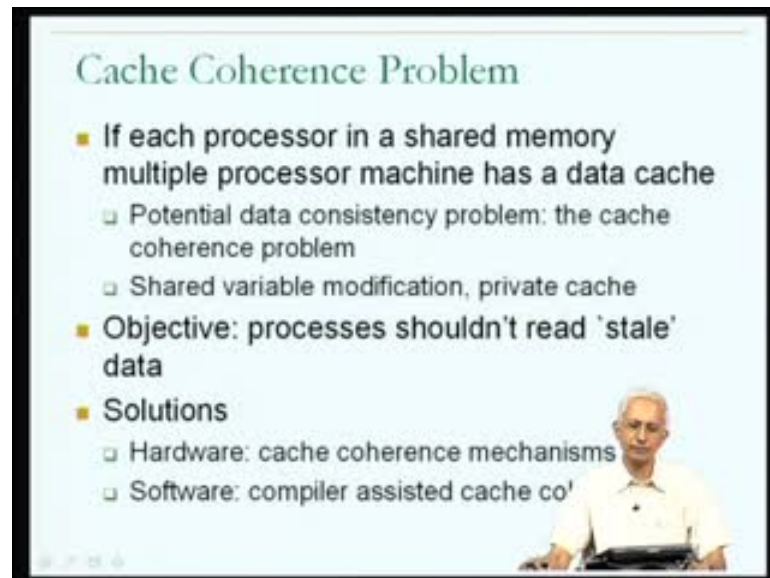
Welcome to lecture 39, the course on High Performance Computing. The previous lecture we started looking at parallel architecture. Our objective is to understand a little bit more about the programming of parallel machines, because parallel machines are fairly wide spread today, they provide improved performance, a programs run on parallel machines can run much faster. They also provide the possibility of fall tolerance, which is sometimes an important requirement. In a way, in looking at the possibility of a parallel machine in which the many processors actually share a single physical address space with a single shared memory.

In the previous class we understood that, there is a potential problem if each of the processors has a cache, and this was referred to as the cache coherence problem, which we looked at from an example of understanding what happens when a simple program which has two processes runs on a parallel machine, in which there are four processors, in which each of them has a cache, each of the processors have the cache.

And in the example which we went through, you will notice that there was a parallel program running as two processes, process p 1 and process p 2, they have a shared variable called x, x is initially equal to 0 which is in the main memory. Now, when process p 1 reads x, a copy of the variable x comes into the cache of the processor running process p 1, if the slides were shown, then they could see what was being said. Then subsequently, if process p 1 reads x again, it will get the value of x out of its cache. Now the problem is that, process p 2 could be sharing the variable x, and therefore, if process p 2 also reads x, then copy of x will come into its cache, subsequently process p 2 could read the value of x again and again. But if now, process p 1 or process p 2, for that example, was to modify the value of the variable x, by ordinary x equal to 1 or some kind of a store instruction on the memory variable x, the effect would be that one of the other cached copies of x could become incorrect, and therefore, if process p 2 actually read the variable of x again, it would get the old invalid value of x equal to 0, whereas currently from the perspective of the parallel program x is equal to 1. This is a serious

problem, because the correct operation of the program has been compromised, and is to refer to as the cache coherence problem, which we described in the next slide.

(Refer Slide Time: 02:35)

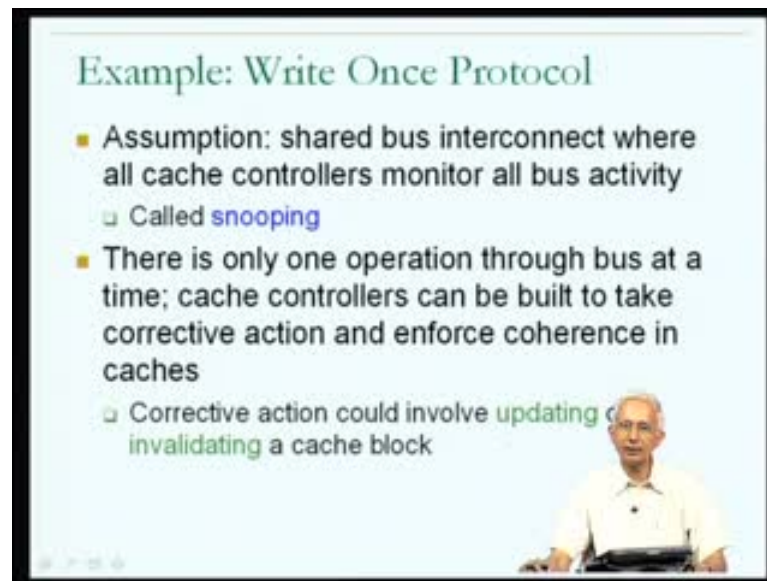


**Cache Coherence Problem**

- If each processor in a shared memory multiple processor machine has a data cache
  - Potential data consistency problem: the cache coherence problem
  - Shared variable modification, private cache
- Objective: processes shouldn't read 'stale' data
- Solutions
  - Hardware: cache coherence mechanisms
  - Software: compiler assisted cache co!

So, if each that the cache coherence problem arose if there is a shared memory parallel machine, in which each of the processors has a cache, and the problem arises, it is a data consistency problem, it arises when there is the need to modify a shared variable, it gets modified in a cache, but maybe not in all the caches. And this has to be handled, it cannot be just left to the programmer to deal with, because it is not possible for the programmer to deal with correction of the incorrect values inside caches, therefore, either the hardware or the system software has to handle this.

(Refer Slide Time: 03:10)

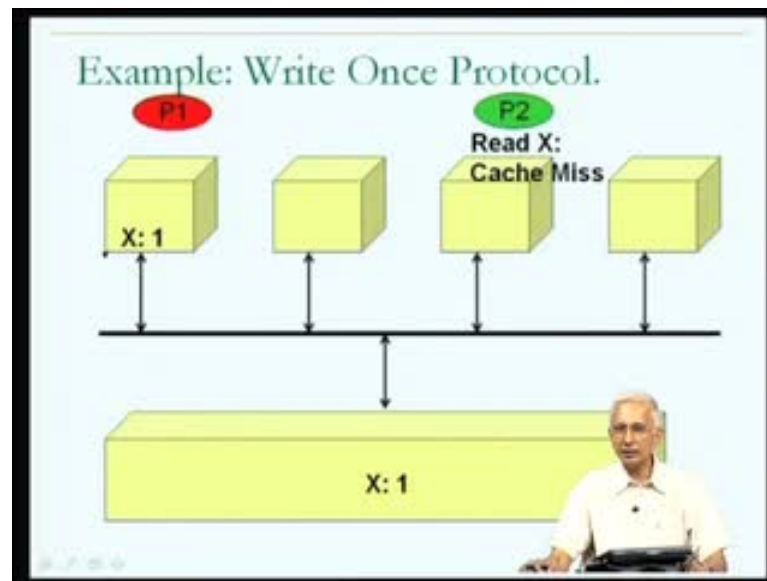


**Example: Write Once Protocol**

- Assumption: shared bus interconnect where all cache controllers monitor all bus activity
  - Called **snooping**
- There is only one operation through bus at a time; cache controllers can be built to take corrective action and enforce coherence in caches
  - Corrective action could involve **updating** or **invalidating** a cache block

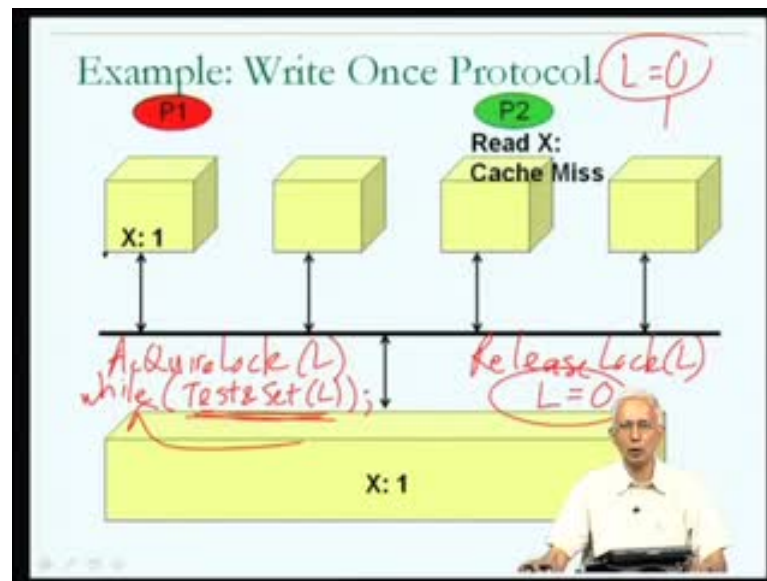
We looked at one simple mechanism, which could be built into shared memory parallel machines, in which there are snooping cache controllers, in other words, in which the cache controller at each of the processors just constantly monitoring what is being done on the shared bus by it and by other processors. So, if this is the case, then whenever there is a an update of some kind, which has happened, or update to main memory, which happens on the shared bus, each of the cache controllers will notice this and can take corrective action, possibly by updating or invalidating its copy of the cache block in question.

(Refer Slide Time: 03:43)



And we saw that this could correct the problem with the previous example, because now when process p 1 reads x and gets a copy of the block containing x, and its cache does not matter if process p 2 gets a copy in its cache, subsequently when process p 1 modifies the value of x under the Write Once Protocol, the copy in its cache will get updated to 1, the copy in the main memory will get updated to 1, the activity of updating main memory will require a shared bus transaction, which will be observed by all the caches in the system, since the cache controllers are snooping. And therefore, each of the other caches in the system, cache controllers could invalidate their copies of the block as shown in the next step, so that if any of the other processes then read the value of x, it would get a cache miss, which is correct behavior as opposed to the previous situation, where it actually read the wrong value of x equal to 0.

(Refer Slide Time: 04:45)



Now, this is a simple idea which can result in the correct execution of programs, but it may not result in good performance for the execution of those programs. As we can, we will realize, if we think about what will happen in a shared memory system which uses something like the Write Once Protocol, a Snoopy Cache Coherence Protocol, but also uses locks to implement mutual exclusion of access to shared variables.

You will remember that, when we talked about concurrent programming, when we are talking about how the operating system shares the CPU among the different processes, we use this idea of a lock, which was a software mechanism built using special instruction, such as a atomic read, modify, write instruction, like test and set. But the lock is required so that, in regions of a concurrent program or a parallel program, where accesses to shared variables are going to take place, it might be important to ensure that at any given point in time only one process of the parallel program is modifying the shared variable at a time.

So, that requirement is still going to be present in parallel programming, if one is using shared variables, and therefore, just like one was using locks to ensure mutual exclusive access to shared variables in a concurrent programming, one would use locks to have mutually exclusive access to shared variables in a parallel program as well, the same concept carries through.

Therefore, in thinking about the Snoopy Cache Coherence Protocol, it does not hurt to also remember that, the processes p 1 and p 2, that we just talked about, will in all likelihood be using locks, to ensure mutual exclusion in their access to the variable x. So, let us now concentrate on the accesses to the locks themselves. So, I am showing the same system that we had before, there are four processors and we have a parallel program running,, but instead of just worrying about the accesses to the shared variables, let us concentrate on what happens to a lock.

So, let us suppose that the value of the lock is initially 0, you will remember that our understanding of locks is; let me just remind you about locks. We are assuming that a lock is some kind of a shared variable, which has a value 0, if the lock is available and have the value of one, if the lock is not available. Further, there are two functions which are provided for manipulating locks; one is a function, which can be used to acquire the lock, and the other is a function, which can be used to release the lock.

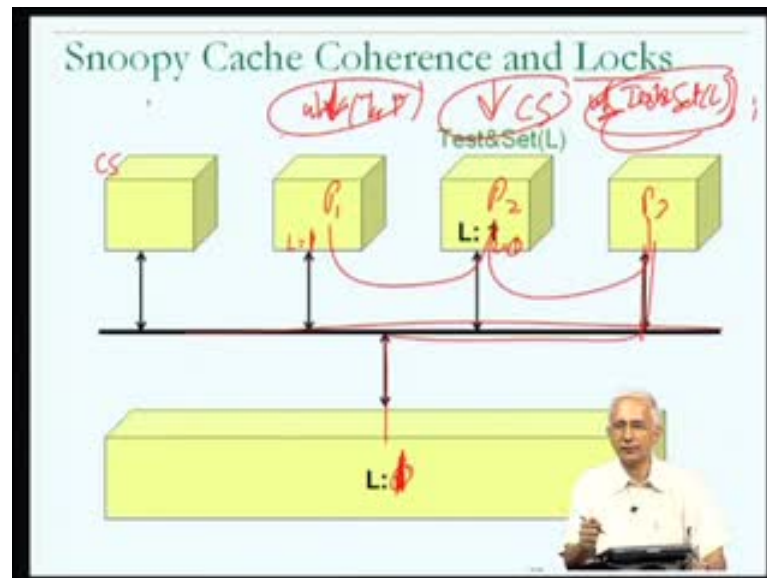
So, if there is a lock call L, a parallel program would, before any of its critical sections, would include a call to acquire lock, and the end of the critical section, would include a call to release lock, and the process would not be allow to enter the critical section, until the lock had been acquired. So, the implementation of release lock was fairly simple, a process just had to set L equal to 0, indicating that the lock is now available.

In implementing acquire lock, we used a special instruction, which is available, there could be different variance on the instruction, but we were using the variant in which there is the instruction called test and set. So, the idea of the implementation of acquired lock was, the process wishing to acquire the lock, would execute a while loop, in which each time through the loop it would test and set the lock variable.

The property of test and set is that, it is an atomic instruction, which will indivisibly do three steps as if they are one step. And the first step is to read the old value of L, second is to modify L to 1, and modify the variable, the memory variable, containing the lock L. So the net effect is that, as long as the process executing acquire lock gets a return value of 1, in other words, the lock is not available, it will continue executing this while loop. But as soon as the value of L becomes equal to 0, because another process has released the lock, then it will get a return value of 0 from test and set of L, and hence might be able to escape from the acquire lock. So, this was how we talk about the implementation

of a lock, and in terms of the parallel program, once again, we assume that the same kind of a lock could be used.

(Refer Slide Time: 08:37)



So, if there is a situation where there is a lock variable, which is initially equal to 0. Suppose that, a process running on the left most processor initially tries to acquire the lock by executing the test and set instruction, since the value of the lock variable is equal to 0, which means that the lock is available, this process which executes test and set of L, will successfully get a return value of 0 and escape from the test and set L loop within its acquired lock function, and therefore, it will acquire the lock setting L to one. So, when it sets L to 1, the cached copy of L gets set equal to 1 and the main memory copy of L also gets set equal to 1.

Now, let us suppose that while this process, the process running on the left most processor is now executing inside its critical section, let us suppose that,, one of a process running from the same program now running on one of the other processors tries to acquire the lock. So, while the process in the left most processor is holding the lock and inside its critical section, let us suppose that a process running on the second processor attempts to do a test in set of L. What happens when it executes test and set of L? In order to test L, it has to have a cache copy of the lock variable, so it gets a copy of L into its cache. And you will notice that, test L sees that the value of L is equal to 1, and

then, indivisibly as the same operation sets L equal to 1, which is why the cache copy inside the left most processor suddenly disappeared.

Now, let me just repeat this once to make sure it clearly understood, when the test and set L instruction is executed on the second processor, they will be a cache miss, and therefore, a copy of the cache of the variable L will be brought to the block containing, the cache containing of that particular processor.

But, indivisibly it is going to set L to 1, and that is going to be a modification of L, and therefore, under the Cache Coherence Protocol, the main memory copy will also get set to 1, and because there is a bus activity, the cache copy inside the other processor will get invalidated, flushed, which is why we now have a situation that looks like this.

So, when the process on the second processors execute a test and set of L, it caused the copy of the lock in the other cache to get invalidated, but it itself did not get the lock, because remember, that the return value from the execution of test and set of L on the second processor was that the value of L is equal to 1, L is not available, and therefore, it was not successful.

Now, let us suppose at this point, a process running on the third processor executes test and set of L. Once again it will suffer a cache miss, it will bring a copy of L into its cache, it will then set L indivisibly as part of the same operation, and the net effect is going to be that the main memory copy gets updated by an activity on the bus, and the cache copy in the second processor gets invalidated. Now, there is one valid copy of the cache block containing the lock L, but that is in the third processor.

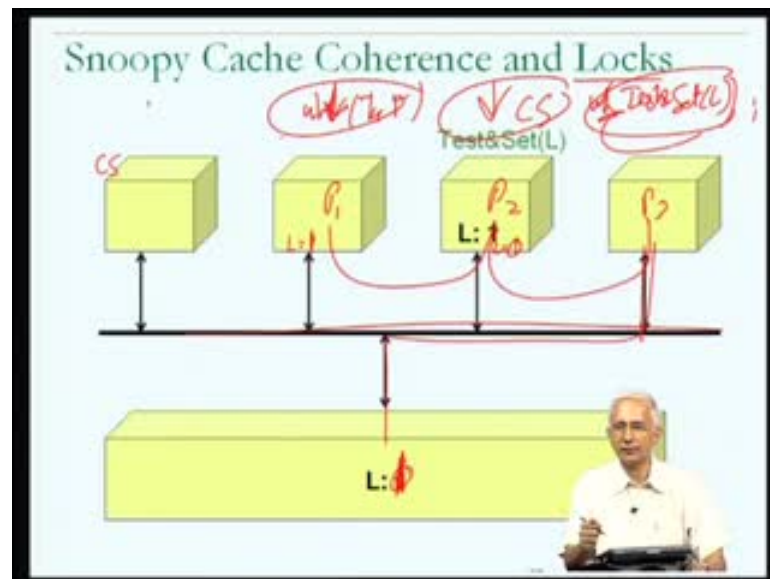
Now, as you can see, when any one of processes executing on any of the processors executes test and set of L, for example, now the fourth, a process running on the fourth processors are executes test and set of L, something very similar is going to happen. The cache copy inside the third processor will get invalidated, because of the update of the main memory copy of the lock L inside due to the test and set instruction being executed, in effect, what is going to happen is, the lock variable is going to move from cache to cache among all the different processors which have a process trying to acquire the lock.



Ultimately, the process which was executing in the critical section is going to release the lock, and that is going to happen, well again, as each of the processes executes test and set of L, it will cause the copy of the block containing the lock to enter its cache, and it will involve the memory operation, it will involve use of the bus.

But soon or later, the process which is inside the critical section, will execute the release lock function, which will involve setting L equal to 0, as a result of which L will be 0 in its cache, L will be 0 in the main memory, and all the cache copies of L in other caches will disappear. Subsequently, the first process among the remaining processors which was trying to test and set the lock, let us suppose, it is now the third processor in line executes test and set of L, it will get copy in its block, which will have a set, initially have a value L equal to 0, it will then test and set it, as a result of which L becomes equal to 1 and other cache copy disappear, the main memory copy will become equal to 1, and it can then enter the critical section.

(Refer Slide Time: 08:37)



Now, one can see that the operation of the lock is correct. The Cache Coherence has correctly ensured that the different processors, the different processes running on the different processors, do not incorrectly update the value of the variable L, which happens to be a lock, and that the mutual exclusion is in fact, guaranteed. It was not the case, that a processor, a process was able to enter a critical section despite not having the lock.

The function of the program is entirely correct, but if you think about things more critically, when I had a parallel program in which each of the of the three processes was executing a while loop, in which it was doing test and set of L, which is what was happening in our example, the process running on this processor; process p 1, processor p 2 and processor p 3; were all executing the acquire lock function and that acquire lock function, they would be executing test and set of L repeatedly, a single instruction inside a while loop. And therefore, the lock variable was actually going to move from one cache to another cache with each execution of test and set of L on any of the caches, and each of those operations involve a bus transaction and updating of main memory.

And this was happening quite frequently, because remember, each of the processes running on the processors p 1 through p 3 was executing the acquire lock function, in which there was a very small while loop, in which it executed a test in set of L and then executed test and set of L again, until the exit condition was reached.

Therefore, if these are processors are running on a one gigahertz clock, then they are going to be executing test and set of L many million times per, I mean, going to executing test and set of L once every 2 or 3 nanoseconds, because that size of the loop is very small.

(Refer Slide Time: 15:15)



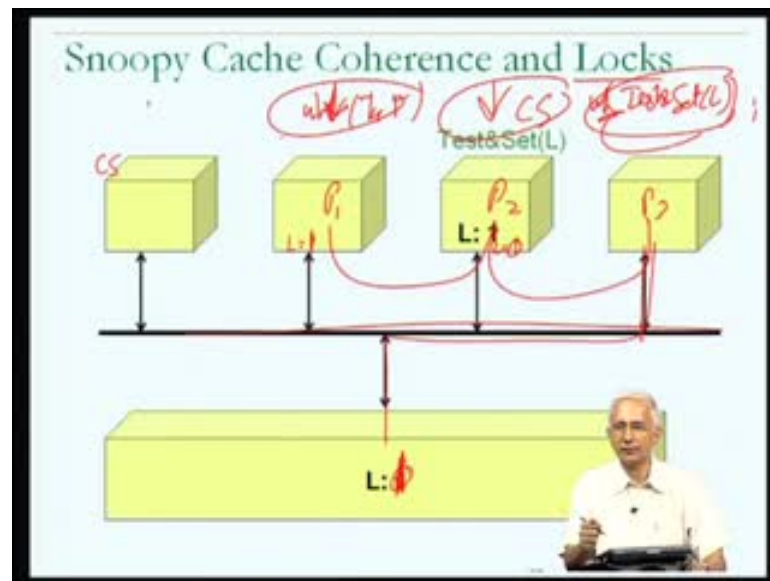
**Lock Implementation**

```
while ( Test&Set (L) );
```

- With snoopy invalidate cache coherence protocol, spinning on Test&Set leads to lock pingponging

Therefore, the net effect is going to be, that if we use this implementation of the lock acquire lock function on a shared memory machine in which there are caches and Cache Coherence is in first using a snoopy invalidate Cache Coherence Protocol, like the Write Once Protocol, then we are going to have a new problem that gets created. The problem, of the lock actually moving back and forth, from cache to cache, at a very high frequency.

(Refer Slide Time: 08:37)



(Refer Slide Time: 16:25)

**Lock Implementation**

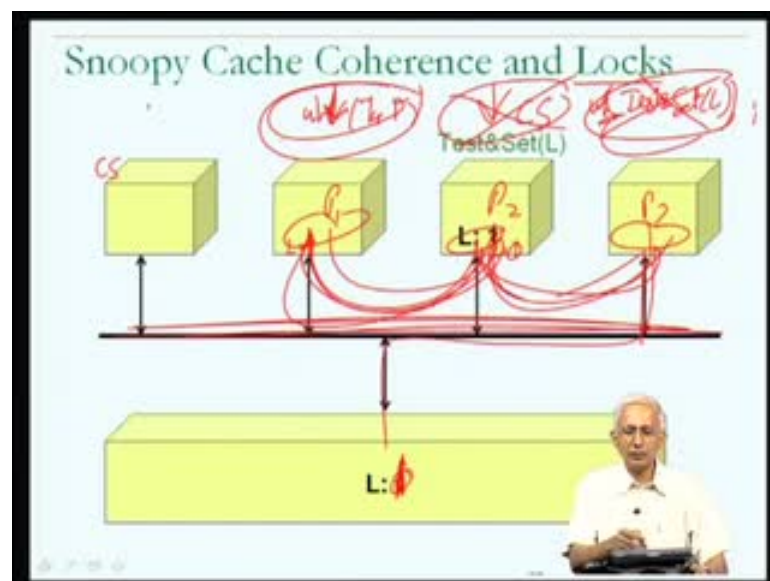
```
while ( Test&Set (L) );
```

- With snoopy invalidate cache coherence protocol, spinning on Test&Set leads to lock pingponging
- High bus utilization slows down memory accesses

So, as I have mentioned, once every few nanoseconds, one or the other of these processors is going to be executing test and set of L as it spins on the busy way it lock loop, and every time any one of them executes test and set of L, the block containing the lock variable is going to move from cache to cache, updating main memory along the way. And this has been refer to by some people as, pingponging of the lock. You can think of it as pingponging, in the sense that, if you think of the lock as being a ping pong ball, then it is moving at a very high frequency, in the game of table tennis or ping pong, the ball moves at very high frequency from the 2 half of the table. In this case, the lock is going to move at very frequency from cache to cache, among all the caches, which are busy waiting on the lock. And hence, the term pingponging is not inappropriate, it is going to be moving at very high speed. If the players of the table tennis game are very skillful, the ping pong, the table tennis ball moves with very high frequency from side to side. And the net effect is going to be that, the performance of the entire computer system is going to suffer.

The correctness of the individual programs is not at risk here, because we have seen that the Cache Coherence Protocol is enforcing the correct operation of the lock, the correct access to the shared variables, that is not a problem. But what has been created through this artifact of the Cache Coherence Protocol is that; the bus, the shared bus and the main memory are going to be kept busy with these test and set instructions.

(Refer Slide Time: 08:37)



So, the bus utilization is going to be extremely high, because once every few nanoseconds is, I just pointed out, conceivably, a processor is going to initiate a bus action to update main memory and its attempt to acquire the lock. Therefore, the bus utilization is going to be so high that the bus is rarely going to be available for any other processor in this system to utilize the bus, and that in effect, even if I had talked about an example, in which there was only one process, which was doing test and set of L, let us assume that, process p1, p2 and process p3 were not doing test and set of L, even if only process p1 was doing test and set of L, the effect might be as bad, the bus may be fully utilized in just the activity of the busy waiting on the lock by one or two processors.

(Refer Slide Time: 17:48)

**Lock Implementation**

`while ( Test&Set (L) );` ←

With snoopy invalidate cache coherence protocol, spinning on Test&Set leads to lock pingponging

High bus utilization slows down memory accesses

*AcquireLock(L):*

repeat

`while (L)` — L:=0

`until ( ! Test&Set (L) );` ←

- Reads of L will be cache hits – no bus traffic
- When lock is available, many spinners may find that L=0. First one to get Test&Set on bus wins and causes invalidation of other cache copies

So, the primary problem that we have is, we have a situation where we needed locks for the correct implementation of mutual exclusion for the critical sections of the parallel program which is to be executed, and there was a need for a Cache Coherence Protocol, in order to ensure that the cached copies of the data, including the locks, did not become invalid or staled.

But when you put the two together, you could have a problem that the Cache Coherence Protocol results in a very high bus utilization, leading a possibility that memory cannot be access at all by the other program, other processes running on other processors of the system, or by other programs running on the other processes of the system, and this is the severe problem which effects the system as a whole, as I mentioned.

Therefore, in the context of a shared memory multiprocessor with the Snoopy Cache Coherence Protocol, such as the one that we just saw, but what should one conclude? One should conclude that this might not be the correct way to implement a lock. By implementing a lock using busy waiting on test and set of L, one is causing a severe problem to the efficient execution of the system, one is not causing a problem for the correctness of the programs running on the system, but there was a problem for the efficient use of the resources of the system, the utilization of the resources of the system. So, one has to think about alternative implementations for the acquire lock function in the light of Cache Coherence Protocols.

Now, the problem with this particular implementation is, that it was a busy wait loop, which was busy waiting on the test and set instruction itself. One could actually think of having an implementation of a lock, in which the acquire lock function does not busy wait on the test and set instruction, but rather, busy waits on the value of the lock. So, as long as the value of the lock is equal to 0, it will loop, until the value of the lock becomes equal to 1.

Now, we saw that this was the beginning of a problem when we try to implement a lock using this mechanism, because there was a need for an indivisible read, modify, write, updating of the lock variable, and that when we try to implement the lock using a busy waiting on the value of the lock, we ended up with an incorrect implementation of the lock.

Now, we can overcome that problem by actually using the test and set instruction to do the updating of the lock. So, while we are going to busy wait on a read of the lock, we will actually have an outer line busy wait loop, in which we will busy wait on the test and set of the lock.

So, we will have a repeat until not test and set of L, as the correct, in order to ensure the correct implementation of the lock function itself. Remember, what we are talking about here is how the acquire lock function is going to be implemented. This is the implementation of acquire lock. And the objective of this implementation is to make sure that, the busy waiting on the lock is primarily on the lock itself and not on a test and set of L. Realizing that the reason that the lock ping ponged between the different caches was, because we were busy waiting, and in the busy wait operation we were, every few

nanoseconds, testing and setting the lock which was a modification of the lock. Over here, we overcome that problem by, busy waiting on the value of the lock and not on a operation, where we are modifying the value of the lock.

Now, sooner or later, a process will exit from this loop because L will become equal to 0, when it is reset by another of the processors, and at that point in time it comes to its outer loop, where it will attempt to test and set the lock. Remember, that there could be many processes, as in our previous example, they were three or four processes, all of whom were trying to acquire the lock.

So, it could happen that more than one of the processes could notice that L has become equal to 0. But, we have the safety mechanism that only one of them will actually observe that it has successfully change the lock from the value of 0 to the value of 1, the remaining processes will notice that they have changed the lock of the from 1 to 1, and we will continue to go back and busy wait on the **read of L**.

Therefore, this will continue to operate correctly, it will be a little bit better than the previous implementation, in that the bulk of the processes will be busy waiting on a read of the lock, rather than on a test and set of the lock. And the net effect is going to once again be correct. There is a minor problem with this implementation, in that, as I pointed out, it could be the case that more than one of the processes may notice that the lock has become equal to 0, because we may just execute that check of L at a point in time, soon after the point in time, at which L has become equal to 0.

So, for example, it could be the ten processes, all notice that L has become equal to 0, and that all of them try to test and set L. But only one of them will successfully test and set L, which is why I say, the first one of them to get test and set on the bus wins, and will cause invalidation of the other cache copies. But there is a problem with this implementation, in that, soon after a lock gets released, there will be a lot of bus activity, as in this example, nine or ten other processes all try to test and set the lock, again using a modify instruction, which will cause bus activity.

(Refer Slide Time: 22:56)

Lock Implementation ...

- But, many processes finding  $L=0$  will all try and do  $\text{Test\&Set}(L)$  causing a burst of bus traffic
- Could try and prevent all of these processes from attempting  $\text{Test\&Set}$  at about the same time

```
repeat  
  while (L):  
    wait ( different time for each process )  
  until ( ! Test&Set (L) );
```

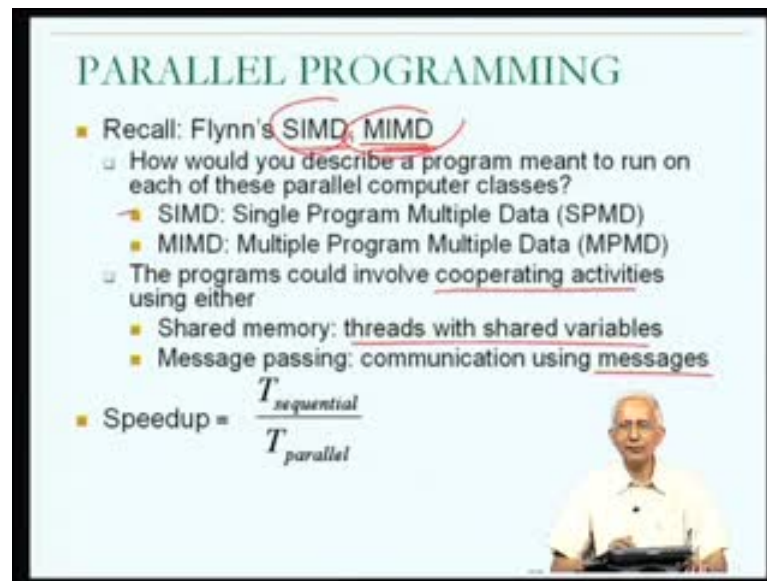
A man in a white shirt is sitting at a desk in the bottom right corner of the slide, with a red arrow pointing from the code snippet to him.

Therefore, one could improve on this particular implementation of the lock even more by setting things up, so that once again, one is busy waiting on a read of the lock, but rather than immediately allowing, after the lock becomes available, allowing all the processes which observe that the lock has become available to busy wait on test and set of  $L$ , which is what we had in the previous implementation, one could actually set things up, so that, between exiting from the this while loop and checking to see whether the lock can be acquired, the different processes could wait for different amounts of time. And this will cause them to actually end up attempting to test and set the lock not at approximately the same time, but at different points in time, and therefore, reducing the intensity of the impact on the bus utilization.

So, as one thinks about things little bit, one realizes that the implementation of the acquire lock function within a system, which is a parallel computer with some kind of a Cache Coherence Protocol along the lines, what we have seen, could be a little bit more complicated than the simple implementation of a lock, that we had talked about earlier. And that the primary objective would be try to make sure, that the correct operation of the lock is ensured, but in addition to that, the efficient operation of the system as a whole is not compromised. So, lock implementations could end up being a little bit more complicated than what we had imagined, from our discussion of concurrent programming. This is in the context of parallel machines.



(Refer Slide Time: 24:19)



**PARALLEL PROGRAMMING**

- Recall: Flynn's SIMD, MIMD
  - How would you describe a program meant to run on each of these parallel computer classes?
    - SIMD: Single Program Multiple Data (SPMD)
    - MIMD: Multiple Program Multiple Data (MPMD)
  - The programs could involve cooperating activities using either
    - Shared memory: threads with shared variables
    - Message passing: communication using messages
- Speedup =  $\frac{T_{sequential}}{T_{parallel}}$

Now, with this we have a rough idea about what to expect in the different kinds of parallel machines. Remember that, there are primarily two kinds of parallel machines; there are the shared memory parallel machines and there are the message passing parallel machines.

We have seen a little bit about shared memory programming before, in the sense that, when we talked about concurrent programming, we were talking about issues, such as the need to have mutual exclusion, the need for locks, the need for shared variables etcetera, all of these issues related to concurrent programming between processes which had shared variables, they also relate to parallel programs of, may be, talking about parallel programs with threads or processes that have shared variables, and therefore, what remains for us to see is, more about the message passing kind of parallel programming.

But before getting to that, I would like to make a few comments about parallel programming in general. Now, you will recall that, when we talked about parallel architecture, I talked about two kinds of classification; one was the classification, where we talked about parallel machines as either being good for shared memory types of parallel programming or a message passing types of parallel programming.

Before that, we talked about Flynn's classification, in which there was the Single Instruction Stream Multiple Data Stream kind of parallel machine and the Multiple Instruction Stream Multiple Data Stream kind of parallel machine. They differ in that, for example, in the Flynn's SIMD kind of parallel machine, at any given point in time there is one instruction which is being executed by all the processors in the system or all the ALUs in the system, in the example that I used, each of them would be operating on a different set of data to different operands.

So, if there were thousand ALUs, each of them could have two different input operands for doing the one operation, that they were all doing at a particular point in time, as indicated by their one instruction. And the alternative was, the Multiple Instruction Multiple Data kind of a scenario, where we could, in fact, think of the as being a single program which runs with one process on each of the processors, alternatively we could think of it as a piece of hardware, on which we could actually have multiple independent programs running on the multiple processors independent of each other, not cooperating in any in any fashion what so ever.

Now given this, one would expect that, if one was writing a program to run on an SIMD machine to probably be different from the program, that one would write to run on a MIMD machine. And it might be good just to talk a little bit about the different kinds of programs, so the different kinds of programming models that might be involved, if at the at the highest level, and thinking about SIMD versus MIMD, for example.

Now, the property of the SIMD, I will remind you again. In an SIMD machine, at any given point in time there is one instruction which is being executed on all the processors of the parallel machine, and therefore, the program itself is going to have parallelism expressed in terms of the different data, which are going to be operated on the different processors by that one instruction. And therefore, one could think about SIMD is having this property of the same program running on each of the processors, but each of the processors using a different piece of data. And one might think about the program, in effect is being a single program multiple data kind of a program, there was the same instruction executing on each of the processors, they just happen to be using different data. Hence the name Single Program Multiple Data.

The alternative is what you might talk of for an MIMD machine, where each of the processors is actually capable of executing a different program, and that you could, in fact, have a Multiple Program Multiple Data kind of a setup to write programs for a MIMD machine. So, one may come across terminologies like this, when it comes to parallel programming.

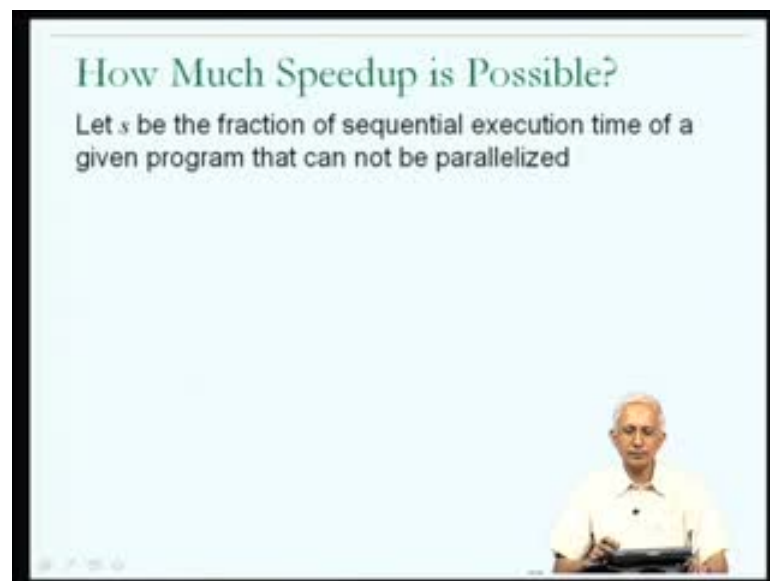
Now, in either of these cases the writing of the programs may involve cooperating activities, and once again I am assuming that I am using the SIMD machine or MIMD machine, I have written a parallel program and that the parallel program from now on is something which has the single objective, and I happen to have written the parallel program as multiple processes, if I have an MIMD machine, I happened to have written it as these instructions which have separate pieces of data, if the SIMD machine. But, in both cases it is a parallel program, in which there is a need for cooperation between the different parallel activities. We have seen that there are these two main mechanisms for the cooperation and they were shared memory and when we talked about concurrent programs, I talked about this as primarily being threads with shared variables, we can also think about processes with shared variables. Then there was the idea of message passing, and just briefly we had talked about message passing as being this mode of interaction between processes, in which there was explicit communication from one process to the other process, using something called, a message, which was supported by functions provided by the operating system.

Now, in any of these cases a question which you may ask is, what is the benefit of running the application as a parallel program. Remember that, we talked about parallel architecture as potentially giving a performance benefit, because it would should take less time for a program to execute, because there is a possibility of having three or if there are  $n$  processors on the system, I could have  $n$  instructions executing at the same time, completing at the same time. And therefore, I could have sufficient improvement in the execution time of the program.

So, we could actually ask the question how many times faster could I expect the program to execute on a parallel machine, and the terms speedup we had seen when we talked about pipelines, we could also use the term speedup in very much the same way to quantify the extent to which parallel programming might benefit us in terms of how much the execution time of a program might reduce. Now, we are essentially going to

compute the execution time of the parallel program, which is the benefit that we would get by running the application on a parallel machine, we will compare that execution time with the execution time of a program if it was running on a single processor. And therefore, we will talk about speedup as being the ratio of the amount of time to execute the application as a sequential or a single process program divided by the amount of time that it takes to execute the application as a parallel program. And, we expect that the denominator is going to be lower because the execution time should come down, if I have written the program to run effectively on the parallel program, and therefore, we should have speedup which are greater than 1. So, in general when we talk about speedup we talk about that ratio between the execution time on one processor divided by the execution time on the parallel machine.

(Refer Slide Time: 30:48)



So, the first question which will come to mind is, if I have a parallel machine with  $n$  processors on it, what is the maximum possible speedup that I should be able to get? Very clearly, the speedup that I could get is going to depend on how effectively I write the parallel version of the program, how effectively I setup the communication between the parallel activities and so on. But, here we are trying to think about the issue of speedup in the limiting sense.

(Refer Slide Time: 31:23)

**PARALLEL PROGRAMMING**

- Recall: Flynn's SIMD, MIMD
  - How would you describe a program meant to run on each of these parallel computer classes?
    - SIMD: Single Program Multiple Data (SPMD)
    - MIMD: Multiple Program Multiple Data (MPMD)
  - The programs could involve cooperating activities using either
    - Shared memory: threads with shared variables
    - Message passing: communication using messages
- Speedup =  $\frac{T_{\text{sequential}}}{T_{\text{parallel}}}$ 

*Sequential program*  
↓  
*parallel program*

Now, we are going to assume that I have a situation where I have a processor with, I have a parallel machine with n processors, but let us make some finer assumptions, we realize that if I have a situation program which was written to run sequentially, it might be the case that I take that sequential program, and then modify the sequential program to get a parallel program. In other words, identify the different activities which can be done in parallel, and then I cause them to execute as separate threads or a separate processes and so on.

(Refer Slide Time: 31:50)

**How Much Speedup is Possible?**

Let  $s$  be the fraction of sequential execution time of a given program that can not be parallelized

Assume that program is parallelized so that the remaining part  $(1 - s)$  is perfectly divided to run in parallel across  $n$  processors

$\lim_{n \rightarrow \infty} \text{Speedup} = \frac{1}{s}$

*10*


*$s = 0.10$     $1-s = 0.90$*

*$1-s \rightarrow 0$*

*2n*

**i.e., the maximum speedup achievable is limited by the sequential fraction of the sequential program**

**Amdahl's Law**



So, one could view the activity as happening in this fashion. We went from a sequential program to the parallel program, and let us suppose that in looking at the sequential program, I realized that there was some parts of the sequential program, which just could not be parallelized, there was no scope for more than one activity happening at a time. Let us suppose that I identify that the fraction of the sequential execution time which was of that kind was  $s$ , so let me just explain what I mean here. Let us suppose that, I looked at a sequential program and I figure out that 10 percent of the parallel of the sequential program could not be parallelized at all, then I might say that  $s$  for that parallel program is equal to 10 percent or point 1. So, we are going to assume that, if I analyze the sequential version of the program, I am able to estimate, I am able to find out what fraction of the sequential execution time cannot be parallelized, by understanding of the work that is being done in that region of the program. What this means is that, the remaining  $1 - s$  or, in this example, 90 percent or point 9 can be parallelized.

Now, let me assume that, if  $1 - s$  or, in this example, 90 percent of the parallel program can be parallelized, then I will make the assumption that, when I divide that activity across the  $n$  processors of the parallel system, I am going to assume that it gets parallelized perfectly. In other words, the time which was  $1 - s$  will actually get equally divided among the  $n$  processors and get parallelized ideally.

Now, with these two assumptions I will actually be able to calculate what the speedup could be. Now, what is the sequential execution time? The sequential execution time is going to be the sum of the fraction that could not be parallelized plus the fraction that could be parallelized. So, that is going to be equal to 1, because here we have reduced the execution times to these two components, each of which is a fraction, the sum of these two fractions is going to be equal to 1. How do I calculate the parallel execution time? The way that I calculate the parallel execution time is, I realize that of the sequential execution time, the fraction which I called  $s$ , in this example, it was point one could not be parallelized. And therefore, in the parallel program this is still going to take the time fraction  $s$ . However, the remaining time  $1 - s$  is going to get perfectly divided across  $n$  processors, which means that if I looked at the  $n$  processors, each would them at be simultaneously doing the activity which used to take  $1 - s$  fraction of the sequential execution time, but that is now going to get equally divided among the  $n$  processors and that therefore, each of them is going to use  $1 - s$  divided by  $n$

amount of time as a fraction of the original program execution time. Therefore, I could calculate the speedup as,  $1$  divided by  $s$  plus  $1$  minus  $s$  divided by  $n$ ; where the denominator is the perfect execution time on a parallel machine, where I could not parallelize this fraction  $s$ , but I perfectly parallelized the remaining  $1$  minus  $s$ . Therefore, this is the best possible speedup that I could imagine.

Now, in the ideal case if I have an infinite number of processors, how much will the speedup amount to? I can calculate that by looking at the limit as  $n$  tends to infinity. And what is the limit of this expression as  $n$  tends to infinity. This  $1$  minus  $s$  divided by  $n$  will tend to  $0$ , and I will be left with  $1$  divided by  $s$ . So, this is the answer we were looking for.

So, if you have a sequential program, in which the fraction  $s$  of its sequential execution time cannot be parallelized, and then you took the remaining  $1$  minus  $s$  and perfectly parallelized it across an infinite number of processors, then the best speedup that you could get would, in fact be  $1$  divided by  $s$ , even if you had an infinite number of processors. And what does this tell us? This tells us that, in general, the maximum speed up that can be achieved on a parallel machine is limited by the sequential fraction of the sequential program, remember that,  $s$  was the sequential fraction. When I looked at the sequential execution time, I figured out the 10 percent of the sequential execution time could not be parallelized. It is just that ten percent or 0 point 1, which determines the best possible speedup that I could get on a perfect parallel machine with infinite number of processors.

(Refer Slide Time: 31:50)

**How Much Speedup is Possible?**

Let  $s$  be the fraction of sequential execution time of a given program that can not be parallelized

Assume that program is parallelized so that the remaining part  $(1 - s)$  is perfectly divided to run in parallel across  $n$  processors

$$\lim_{n \rightarrow \infty} \text{Speedup} = \frac{1}{s}$$

|| i.e., the maximum speedup achievable is limited by the sequential fraction of the sequential program

**Amdahl's Law**

This is somewhat negative result, it tells us that even if we have an infinite number of resources and infinitely powerful parallel machine, the speedup that we could get would be limited by this fraction. And in effect, if you work out the maximum speed up that I could get for  $s$  equal to point 1, then you will see that it is a very disappointing speedup of ten. In other words, if I ran this particular parallel program on a machine which has twenty thousand processors, the best speedup that I could get would be 10, it could become 10 times as fast as the sequential program, even if I ran it on a parallel machine with ten thousand processors.

So, this is to some extent, to be viewed as a negative result, but it is to be viewed as a realistic way of looking at things. This particular formulation of what speedup might be is known as Amdahl's law and gives us, it may not actually give us a good idea about what may happen to parallel programs that we write, when we write them to parallel machines, because it may be possible for us to improve the way that the sequential program did the work, even that that we are going to have to do the same work in a parallel machine, and we may be able to get improve speedup, so what this is predicting because this is working under certain assumptions.

But, it does tell us that trying to work with as low a fraction of  $s$  is possible may be to our benefit. In other words, trying to view the application in such a way that, the bulk of it can be parallelized to some extent or the other, rather than, being completely un



parallelizable. So, Amdahl's law allows us to get some kind of a feel for the importance of the sequential part of activity as far as writing a parallel application is concerned.

(Refer Slide Time: 37:38)

**Programming with Message Passing**

- Need
  - 1. Mechanism to create processes to execute on different processors
  - 2. Mechanism to send/receive message
- Must specify the identity of the communicating process
- Example
  - P1: `send(&x, P2)` → P2: `receive(&y, P1)`
- Message passing libraries
  - PVM (1980s)
  - MPI (1990s)

Handwritten annotations: P1 → P2 → P3 → P4; pid 13 15; receive; P1: send(&x, P2) → P2: receive(&y, P1)

Now, our next objective, now that since a little bit about programming with shared variables, and I will remind you that we learned a little bit about programming in shared variables when we talked about concurrent programs, we realize that problems such as , the need to look at regions of a parallel program which have shared variables and modify those shared variables as critical sections and ensuring mutual exclusion in those critical sections, the importance of locks etcetera. We have learned about that, when we talked about concurrent programs, and the principles follow through into parallel programming We have also seen that, while the principles follow through into parallel programming and parallel architecture, the implementation of some of the primitives such as, locks, may have to change, and we saw that in certain kinds of parallel machines, it would not be a good idea to have the same implementation of lock, that we assumed for concurrent programming on a machine which has one processor.

So, we know something about programming with a shared memory. We do need to know something about programming with message passing, which is what we will proceed to do. So, the idea that we understand about message passing is that: message passing is some kind of a facility provided by the operating system for explicit communication of data values from one process to another process. Therefore, we suspect that, what is

needed in order to do parallel program with message passing is, first of all we will going to have to have some kind of a mechanism to create processes to execute on different processors, and we have not talked about this before. But, if I have a system in which there are a thousand processors, and I want to run a parallel program which runs as one process on each processor, it may not be convenient for me to assume that I actually initiate the execution of the program on each of those thousand processors, because for me to type `a dot out` on each of thousand processors is going to take a fairly large amount of time. Therefore, we probably need to assume that a mechanism to create the processes to execute on different processors should be made available. This is not something that we would want to do by hand, even to execute a program `a dot out` on a thousand processors may not be convenient for us to.

Now, second thing that we are going to need is help from the operating system, we need some kind of mechanisms provided by the operating system to send and receive messages. So, there has to be a collection of operating system provided mechanisms; one to send a message, one to receive a message. Now we understood that, send and receive, are the names for the functions provided by operating system, at the sending end to send a message, at the receiving end to receive a message explicitly from one process to another process. So, it seems obvious that, the identity of the communicating processes must be explicitly specified, and that in the send function it must be possible to specify which process one wants the message to be sent to.

As a simple example, it is possible that the operating systems sets things up so that, we refer to the processes by their process IDs. We know that in Linux or UNIX systems, each process has a process ID, which is a small integer and each process on a system will currently have a unique ID.

So, it is possible that process ID of the one process is 13, another process ID of another process is 15, and I am referring to those as P1 and P2 here, but here, we are talking about small integers. Now in general, this may have been for assumption on a system where there is a single operating system and a single processor, but remember, now that we are talking about a situation where we have multiple processors, and it is conceivable let me need to view this as, for example, a situation where there is Linux running on each of those processors, and therefore, it is no longer makes complete sense to talk about the process ID as the Linux process ID associated with the process. Why? Because, I could

have a situation where process 13 running on processor 1, has to communicate with process 15 running on processor 3, and that therefore, just thinking of the process ID as being the Linux process ID may not generalize anymore, we may need to have something a little bit more complicated than that. For the moment, I will just assume that we have that mechanism, the mechanism by which we can identify communicating processes uniquely on any processor of the system.

(Refer Slide Time: 37:38)

**Programming with Message Passing**

- Need
  - 1. Mechanism to create processes to execute on different processors
  - 2. Mechanism to send/receive message
- Must specify the identity of the communicating process
- Example
  - P1: send(&x, P2) → P2: receive(&y, P1)
- Message passing libraries
  - PVM (1980s)
  - MPI (1990s)

*Handwritten annotations:* P1 → P2 → P3 → P4; pid 13 15; receive; red underlines on example code.

But, we expect that what the send function or the send system call, whatever it may be, is going to look like, something like this. And process P1, if it wants to send a piece of data to process P2, will use send, it must explicitly specify that it wants to send it to P2, and it must explicitly mention the data. In this particular example, it has put the value into a variable called x, and the address of the variable x is what is used by the send function or the send system call.

(Refer Slide Time: 37:38)

**Programming with Message Passing**

- Need
  - 1. Mechanism to create processes to execute on different processors
  - 2. Mechanism to send/receive message
- Must specify the identity of the communicating process
- Example
  - P1: `send(&x, P2)`
  - P2: `receive(&y, P1)`
- Message passing libraries
  - PVM (1980s)
  - MPI (1990s)

Handwritten annotations: P1, P2, P3, P4 with arrows; pid 13, 15; receive.

Now, at the receiving end, I am also assuming that the process P 2, which is expecting to receive a value from process P 1, must explicitly execute a receive function or receive system call, depending on what it is, in which it explicitly mentions that it which is to receive a message from process P 1. Why is it necessary for process P 2 to explicitly mention that it wants to receive a message from process P 1? Now, the way to view this is that, in general, when we write parallel programs using message passing, there could be more than two processes communicating, and it could be that there is a need for process P 1 to communicate a process P 2, and that possibly may be as another part of the activity of the parallel program, for process P 3 to communicate with process P 2.

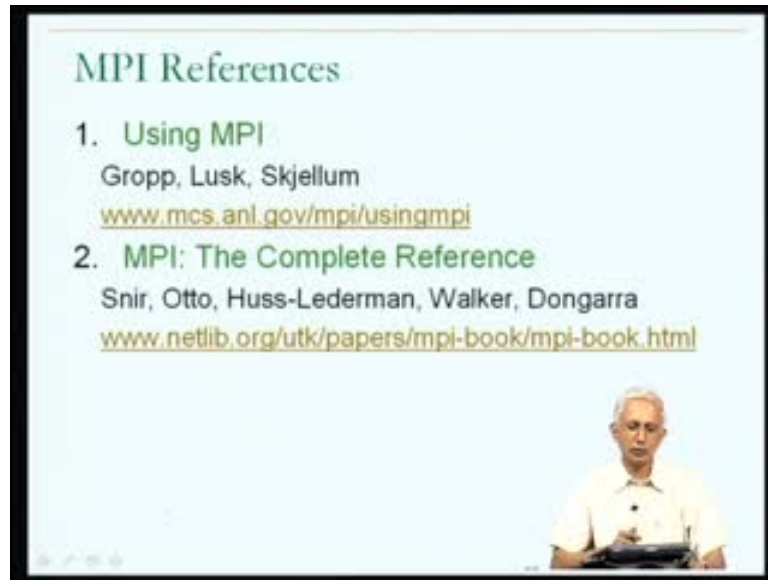
Therefore, within process P 2 they are going to be separate calls, one for the receive from process P1, and one for the receive from process P2. And therefore, it seems necessary to actually have in each of these two calls, in order to distinguish one from the other, explicit mention of which was the sender from which the message is expected.

Once again, the data from process that was send by process P 1, will be received as an activity of the receive function execution, and will be put into a variable of process P 2. So, y is a declared variable of process P 2, and the address of y is what is passed as a parameter to the receive function. So, you will notice that this is a setup for communication between process P 1 and process P 2, in which there are no shared variables between process P1 and process P2, all that is necessary is a mechanism for

sending and receiving messages between processes, and some mechanism for creating these processes on different processors, an addition to that, mechanism to specify the identity of another process. P 1 identifying process P2 as it is communication partner, and process P2 identifying process P1 as it is communication partner.

Now, different operating systems, we provide support for message passing with different sets of functions. And therefore, it is a little difficult for us to understand how we could write a program, a parallel program, which is going to run on, lets say, a thousand processors, in which some of the processors might be running one version of Linux and some of the other processors might be running some other version of UNIX and so on. And might be more productive for us to try to think about a mechanism which is abstracted out above the operating system, and that is what we will assume, we will assume that, in doing message passing programming, we might not be using the operating system provided functions directly, such as, the send and this receive which I have been talking about, but rather, we might use some kind of a library, which will make the appropriate operating system calls as part of its activity, but will make it unnecessary for the programmer to have to know about the nature of the operating system support for message passing on each of the different variants of Linux or each of the different variants of UNIX, that the parallel program may have to run on. And therefore, we expect that there is going to be some kind of this abstraction provided by a library, and if one looks back in time one will find out that a different points in the history of message passing, people have created new libraries for this purpose. If you look back into the 1980s, there was a library which was known as PVM or parallel virtual machine, which used to be fairly popular. Since the 90s, the dominant message passing library has been something called MPI, the library was created sometime in the 90s, and continuous to be a widely used library for message passing programming. And we will therefore, proceed to talk about message passing programming in the light of MPI.

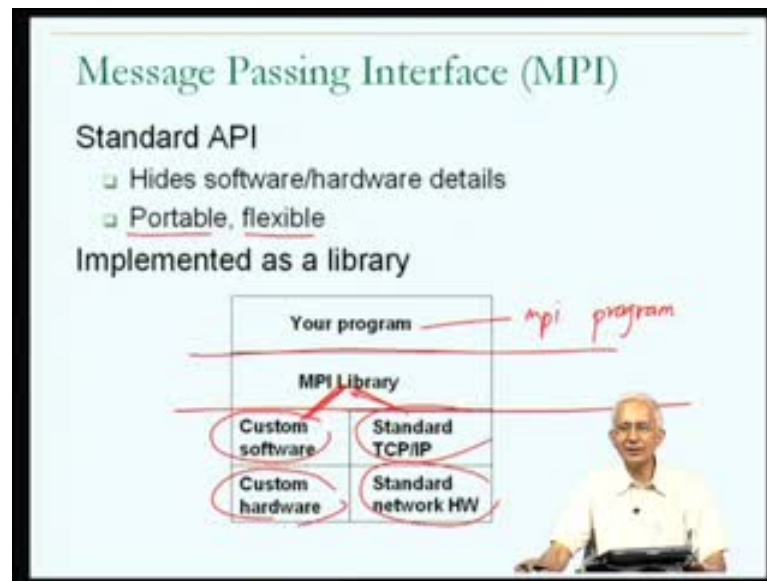
(Refer Slide Time: 46:02)



So, I am going to use MPI as the example of how one can do message passing programming. You will find out that it is widely supported on almost any parallel system that you could talk about, if not one can download the MPI libraries and cause them to be installed on the system, parallel systems, which you are dealing with, and write message passing programs using MPI.

In terms of references, I am giving you two good references to learn a lot about MPI, both of them are available, through the URLs which are provided. Second is a book, which can be downloaded, or chapter by chapter referred to across the internet.

(Refer Slide Time: 46:43)



Now, first of all let me mention that MPI stands for Message Passing Interface, not a surprising name for a library that is primarily intended for writing message passing programs, parallel programs. But, in effect like any other library or interface, it provides the standard application programmer interface for doing message passing on system regardless of how the operating system may actually support the send and receive functionalities. So, in effect the MPI hides the hardware and software details of the underlying parallel machine, it is no longer necessary for the programmer to worry about what functions are available on the version of Linux or the version of UNIX, that the parallel machine happens to be running.

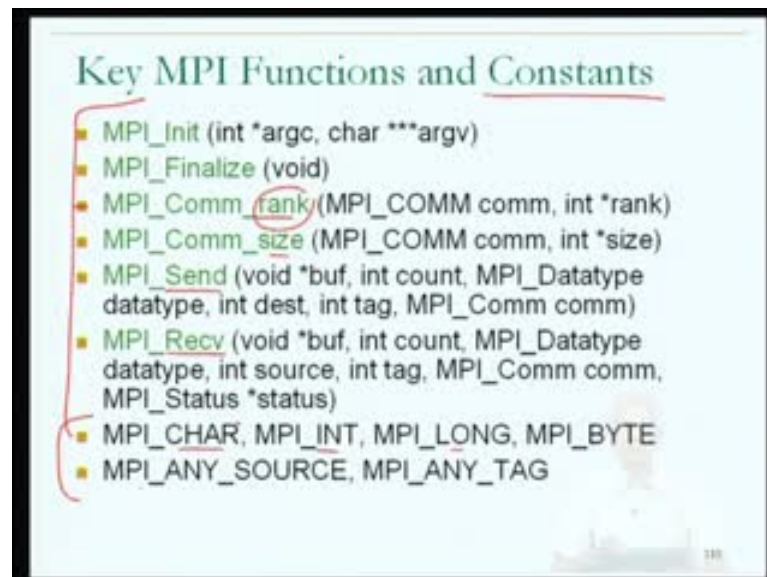
In fact, the programmer need not be aware of the hardware details or the software details. This makes MPI programs portable, in that you could take you could write a parallel application using MPI message passing and run it on one particular parallel machine, you could then take the same MPI program and run it on cluster of network stations and expected to run correctly on the cluster of work stations. So, the portability is good because of the fact that it hides the hardware and software details. There is also me flexibility which results.

Now, as I mentioned, MPI is implemented as a library, it is a collection of functions, it is a collection of include files and the implementation of the functions, which constitute the application programmer interface. From the perspective of your program, once you write

in MPI program, your program that I am talking about over here is, in MPI program that you have written, in other words, the assumption is that you have written a parallel program which is going to run as a collection of processes, and the processes are going to be communicating with each other using message passing, in order to achieve the common objective. And that this is going to be done using the MPI message passing interface functions, not using the operating system provided functions directly.

So, as far as your program is concerned, your program is the top most box over here, your program is written to use the MPI library. Now again, depending on what system you are running your program on, the MPI library may make use of some standard networking functionalities, such as TCP IP functionalities, or it could be making assumptions about some standard network hardware. That is one possibility of what the setup could be on the parallel machine that you are dealing with, the other possibilities that there could be some custom software, some hand crafted networking functionalities, send and receive functionalities, that were created for the system that you are dealing with, that would be call custom software, which because, the hardware was also specially design for that particular system.

(Refer Slide Time: 49:51)



But as far as you as a programmer are concerned, you would just have to know how to deal with MPI. And then, the implementation of MPI would be suitably selected by the person who constructed the system, to be such that, it either used the correct version of



the MPI library, and therefore, your program can be written in this portable flexible frame work. Now as far as we are concerned, MPI then can be viewed as a collection of functions, library of functions. And let me just show you some of the key MPI functions and constants, before we actually go ahead and talk about MPI itself in more detail. I am doing this just to demystify MPI, you will recall that when we talked about system calls, I told you that system call is this very special kind of a part of the operating system and it is a part of the operating system and must be dealt with great respect, one cannot expect that one can executes system calls and modify system calls, but rather I got into it by telling you the names of some of the system calls, and you realize that you had actually use some of those functionalities before in some of your own programs. For the same reason, I am just going to in mention some of the MPI functions, and then we will see that is going to be quite easy to write simple MPI programs. Now the first MPI function that any MPI programmer has to know about is MPI\_Init, and as a name suggest MPI\_Init is a function which must be called once at the beginning of the MPI program to initialize the various activities of the MPI library.

By the same token, you would expect that at the end of the MPI program there is going to be a need to wrap up the various MPI activities, possibly, and therefore, there is going to be a function called the MPI\_Finalize, which must be one of the last things that must be called within the MPI program, to suspect that somewhere in between there is going to be activity of causing processes to run on the different processors of the parallel machine etcetera, and the communication between those processes.

But one would expect to see MPI\_Init at the beginning of the MPI program and MPI finalize towards the end of the MPI program. Now, one of the concepts about MPI that we are going to learn about later is this concept of rank, which will abstract out denotation of the process ID, so rather than talking about the ID of a process from the perspective of the Linux process ID associated with that particular process, we abstract things out into something called an MPI rank. And the number of processes that the program is running as, will be known as the size, and there are functions to determine what the MPI rank of a particular process is, and the process can find out its own rank by calling the MPI function, MPI\_Comm\_rank. I will mention a little more about this later.

Similarly, by calling the function, MPI\_Comm\_size, any given processes of the MPI program can determine how many processes there are in this particular MPI program or

how many processes this particular MPI program is running as. We expect that there are going to be functionalities in MPI for sending and receiving messages, and they go by the names MPI\_Send and MPI\_Receive. There are actually families of functions, I am just showing you representative example, MPI\_Send. You will notice that each of the MPI functions identifies itself by starting with MPI ,underscore and you will notice that there are various parameters to send, various parameters to receive, which we will have to learn more about.

(Refer Slide Time: 49:51)



There are several other MPI functions, but I talked about the fact that there are certain MPI constants. And the MPI constants, once again are declared constants, which are available in the MPI library, and all start with the MPI underscore. And you will notice that, there are some which seem to be used to specify types, for example, MPI\_CHAR is a constant which seems to be used to indicate that a particular variable is of type character. Similarly there is INT, LONG and BYTE. And then, these two mysterious other constants called MPI\_ANY\_SOURCE, MPI\_ANY\_TAG, which we need to understand more about.

So, basically the MPI interface is a collection of functions and a collection of constants. And once one has a understood all the functions and all the constants, one can readily write MPI programs, which means that one can write parallel program, which use message passing for the communication and interaction between the processes, allowing

them to achieve their common objective. In the lecture to follow, we will look at MPI, the MPI functions in more detail, we will understand how they can be used, then we will look at some examples of writing applications as parallel programs using message passing as a mechanism for interaction. We will stop here for today and look at MPI in more detail in the lectures to follow. Thank you.