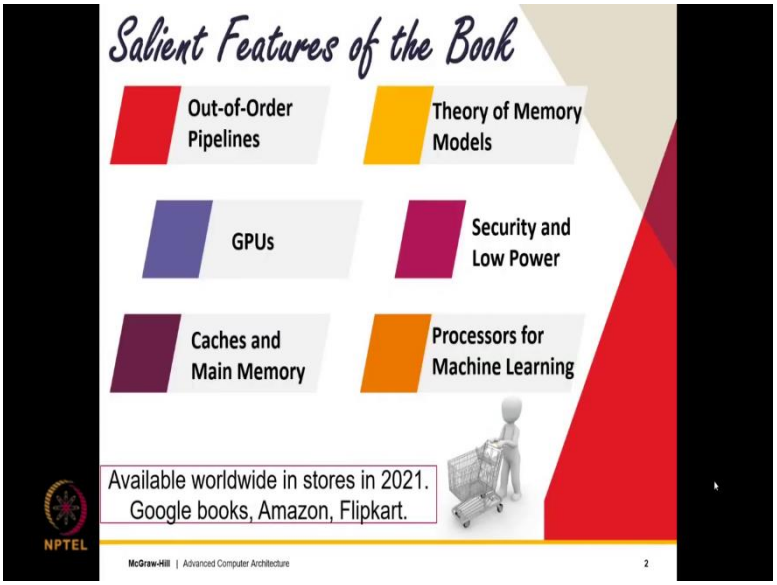
Advanced Computer Architecture
Prof. Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

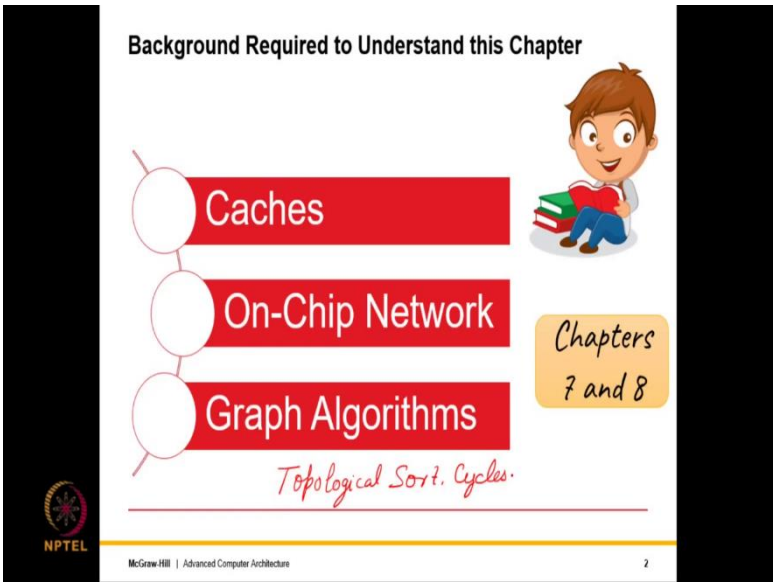
Chapter - 09
Lecture - 25
Multicore Systems Part - I

(Refer Slide Time: 00:23)



So, this is by far the longest chapter in the book and also by far the most complex, but we will break it down into very very simple subchapters and we will discuss each one in great detail; then you will find that this is not that hard after all, because in a certain sense this combines everything that we have learnt up till now.

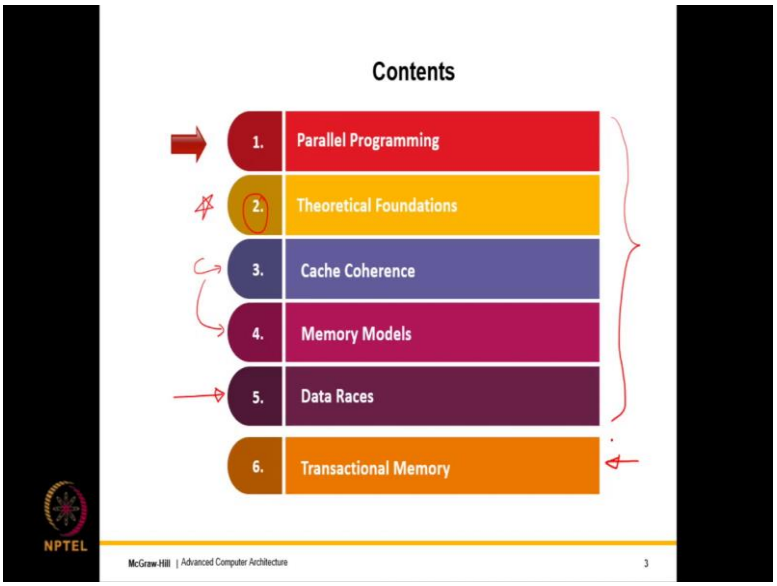
(Refer Slide Time: 00:55)



It combines pipelines, as you will see it combines caches, it combines the on chip network and it also requires a certain amount of knowledge in basic data structures, namely graph algorithms. So, that is required. So, particularly when there are cycles and graphs and also the topological sort algorithm, this is something that students need to know.

So, topological sort existence of cycles in a graph, these are the two basic concepts that students are expected to know and this chapter heavily builds on chapters 7 and 8. So, even if somebody has not gone through chapter 8; but whatever is required, pointers will be supplied at the right points within the chapter.

(Refer Slide Time: 01:49)

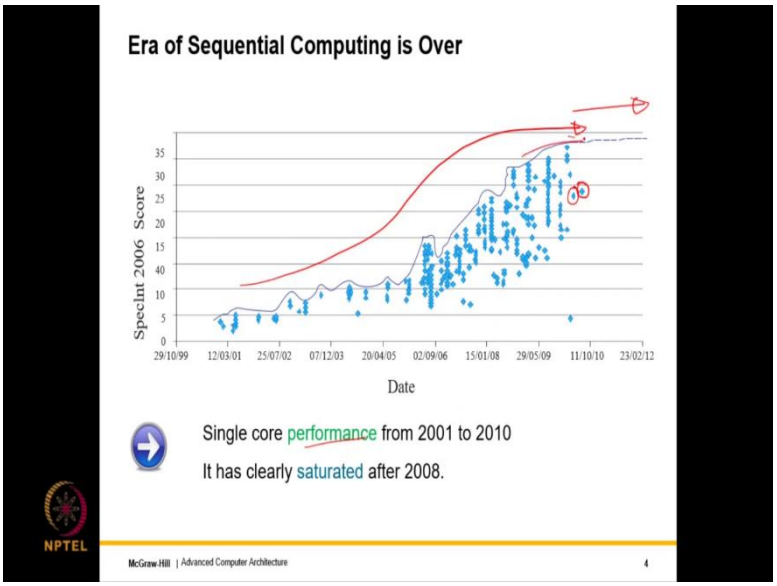


So, this is how we will proceed, the first is that an overview of parallel programming will be provided. So, most of us have written sequential programs throughout our life, we have not written a parallel program; but this entire chapter is about building hardware for parallel programs. The next is a fairly large number of slides on theoretical foundations.

This is very important, because this lays the base for pretty much the entire chapter; then we will discuss cache coherence and memory models. So, if somebody has understood this, then understanding these two is going to be very very easy. And finally, we will discuss data races with a very very important theorem that we will cover over here, which is very important for the correctness of the entire system.

So, if somebody has covered all of these, I would say that they know almost enough to successfully survive and thrive as a multicore engineer. The last module which is on transaction memory is strictly optional. So, if somebody does not have enough time, he or she may not cover this; even though covering it is not discouraged, in the sense if they have enough time, they can look at transactional memory.

(Refer Slide Time: 03:17)



So, first let us motivate the need for multicore systems. So, what I am showing you is some open source data, where the x axis is the date. So, it starts from 1999 to 2012 and the y axis is essentially the performance of processors. The performance represented in the y axis is represented in terms of the specint 2006 score.

So, the specint, these are a set of standard benchmarks distributed by spec the performance evaluation cooperative. So, a specint score is provided. So, where the specint score is essentially it is the ratio of execution time on a given machine with respect to a machine in the past. So, these are relative numbers. So, let us not go in to the unit of the y axis, consider them to be arbitrary units.

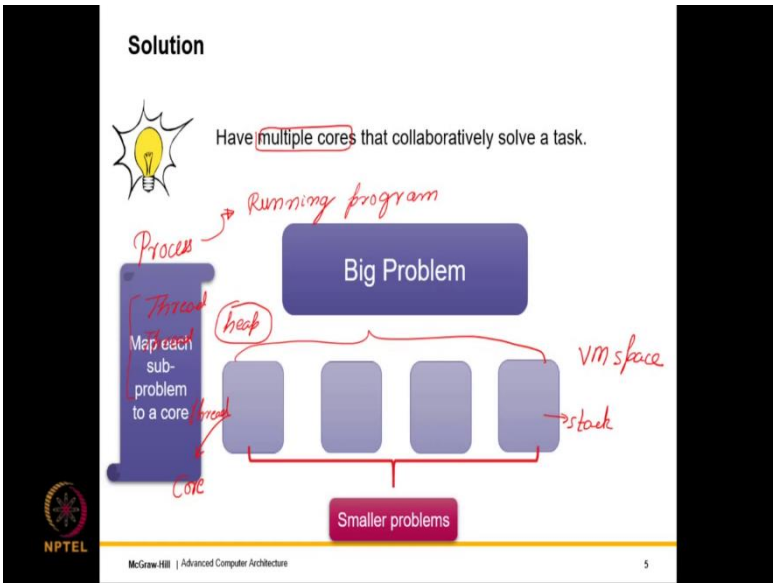
But the most important point to note is that, we are seeing a gradual saturation in single core performance. And this gradual saturation after 2010 or so, so I did not plot; so see each one of these points actually represents a distinct processor, distinct CPU. So, I could have plotted more data, but the results would have been the same; that single core performance by and large has remained constant, it has not changed, it does tend to increase a little bit, primarily because we are able to pack in more cache, because the transistors are getting smaller.

But just because transistors get smaller, it does not really mean that the efficiency is also increasing commensurately. The reason being that there is something called a design rule

in a VLSI fabrication process. So, design rule basically specifies the rules for placing let us say two transistors or two logic gate side by side.

So, what is really happening with design rules is that, they are getting more restrictive, in the sense that it is not really possible to arbitrarily place different kinds of transistors and different kinds of structures that use transistors beside each other; because of that there is a sublinear increase and performance. So, there is a certain amount of saturation. So, as we can see a single core performance has saturated in this time frame, after 2008 you can clearly see the science of saturation.

(Refer Slide Time: 05:59)



So, basically the idea is that instead of a single core, take a large problem, divide it into smaller problems and have multiple cores to collaboratively solve a task. So, what we do is, we take a big problem, divide it into several smaller problems and assign each problem to a separate core. So, each problem is essentially assigned first to a thread. So, a thread is a light weight process. So, I tell you a difference between a process and a thread; that is a very important distinction to make at this level.

Say process is essentially a running instance of a program. These are when you take a program; like let us say if you take power point, so power point as such is a data file, which is residing in the file system, but when I double click it, it becomes a program, a running program. So, a running program, instance of a running program is a process.

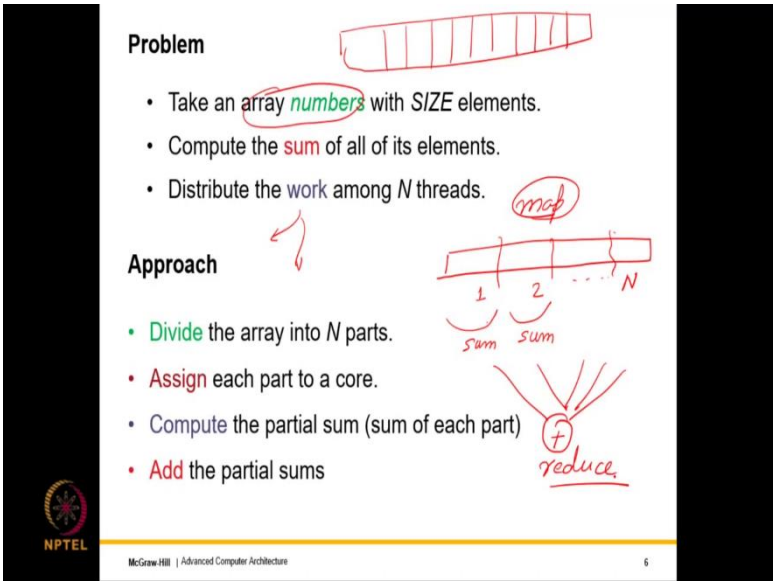
So, a process clearly has its virtual memory and open files network connections all of that. A thread is essentially what is called a lightweight process, in the sense it is a process in its own right; but it we also define a set of sister threads, each thread being a process in its own right, but the threads do share a large part of the virtual address space.

Namely they have their separate stacks and registers that, the that is their private; but other than this private storage, they do share the heap region where all the dynamically allocated arrays go, they do share the core. So, different threads can communicate by reading and writing to shared memory; because they share as I said large parts of the virtual address space.

So, that is important that, these are not really unrelated processes; but instead each of these threads can communicate with each other. So, you can say that they together form a whole. So, Linux define something called a thread group, where instead of one process, I create a thread group with multiple threads; the threads do share as I said a part of the virtual memory space between them. So, what is private to a thread is actually it is stack and of course, it is set of registers and so on, but pretty much that is it.

So, each sub problem is mapped to first a thread in the software space and then to a core. So, that each sub problem is mapped to a separate thread and that thread is map to a core. So, each of these blue boxes over here are essentially cores.

(Refer Slide Time: 09:00)



So, let us consider an example, it is a working example; you will find that the details in the book along with the code. So, I will not explain the exact parallel programming semantics, that is not the aim of the book. So, the aim of the book is to introduce two paradigms to you; one is a shared memory paradigm, other is a message passing paradigm.

So, we will again take some liberty with the syntax. So, the syntax will not be exact; but let us put it this way, it will be similar. So, what is the problem? we take an array numbers which is a very very large array, it can contain a billion elements. So, it contains size elements and it is an array with very large array, it contains size elements.

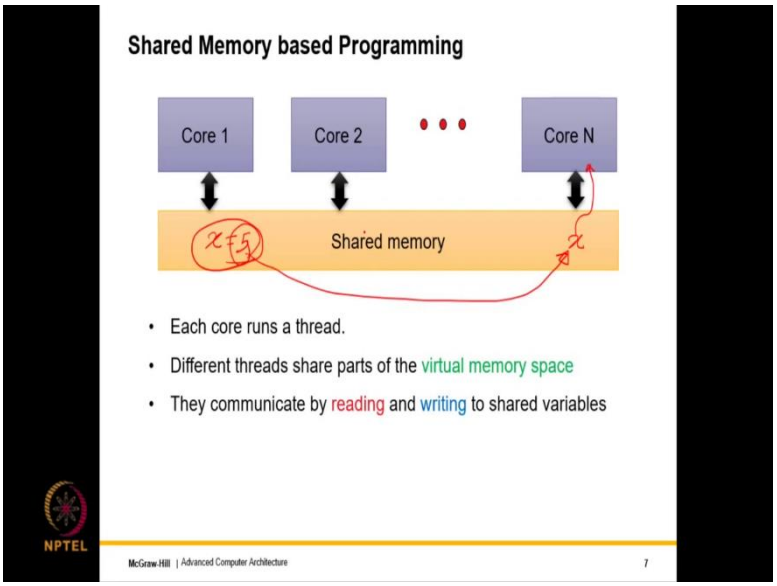
So, the aim is to compute the sum of all of it is elements, that is the aim. So, let us not worry about overflows. So, let us assume that the entire sum does not overflow. So, let us compute the sum of all the elements. So, the way that this is done is that, it is done by distributing the work among the N threads. So, let us distribute them equally.

So, we divide the array into N parts, we assign each part to a core. So, what do we do? We take a large array, we break it down into this is a part 1, part 2, part N; each of these parts is assigned to a core. See we compute the partial sum, which is the sum of each part and then we add the partial sums.

So, each thread computes the sum of each part. So, if there are N threads, we end up with N partial sums, then we add the partial sums. So, this first two operations of actually dividing a large problem into smaller sub problems; the way we have partitioned this array is known as the map step and then each map step produces each step.

Each step over here produces a partial sum and the partial sums are added and this process of addition of the partial sums is known as the reduced step. So, you must have heard of the popular programming paradigm called map reduce. So, map reduce is basically this.

(Refer Slide Time: 11:45)

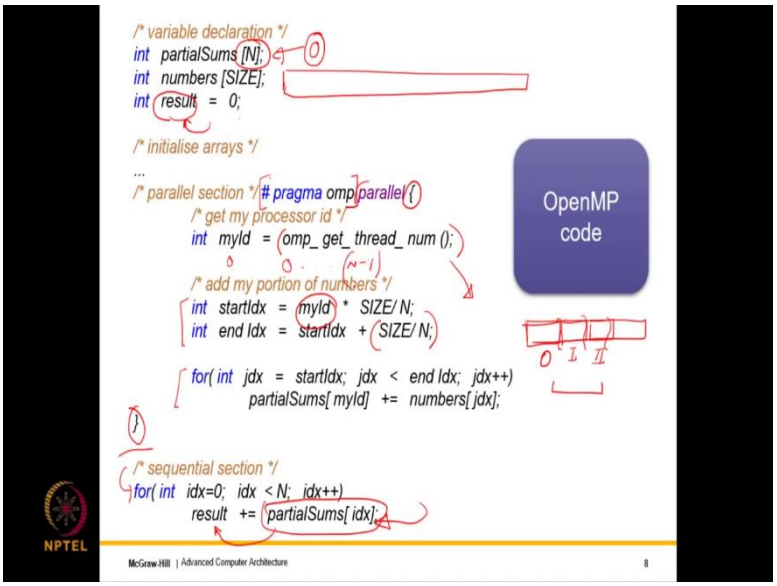


So, let us now look at the first programming paradigm, which is shared memory based paradigm. So, in this case we have N cores and they communicate by a shared memory. So, the way they communicate is that, let us say there is the same variable. So, variable x is let us say assign 5 over here; then we read the variable x into a register and it will automatically get the value 5, subject to the fact that it is being read later.

So, this will of course, be a topic of intense discussion when you something written, when you something read; but for the time being let us assume that all of these things happen, the way that we would expect in an intuitive manner what is intuitive or not we have 50 slides to explain that. So, each core runs a thread, different threads share parts of the virtual memory space and the main communication happens by a reading and writing to shared variables.

So, these shared variables are variables that are not declared on the stack. So, they are typically declared on the heap. So, they are essentially, let us put it this way. Variables that have a global scope, they do not have a function scope; that have a broadly speaking global scope, global to at least the piece of core that the threads are running. So, when we see an example, it will become clear.

(Refer Slide Time: 13:14)



So, let us look at the code. So, this code is the OpenMP code. So, I am not showing all of it, but I am showing the relevant parts of it. So, OpenMP is basically an extension to C C + +; almost all C C + + compilers support OpenMP. So, if you write it an OpenMP which is just a small addition on top of it; they will compile it to parallel code automatically, you do not have to do anything.

So, let us see. So, given we have N threads, so which let us say are defined as # defined constants. So, we have an array of partial sums, where each entry stores a partial sum, int numbers with size entries is the big long array whose sum of sum we want to compute and the final sum will be stored in this variable result.

So, the final sum of adding up all the elements is going to be stored over here. So, the parallel section this is how it is going to look that, we will define our OpenMP directive hash pragma omp. So, # omp for OpenMP and parallel. So, this indicates that whatever is there inside the left curly bracket and the right curly bracket, which is essentially all of this piece of code. This piece of code will execute in parallel.

So, if we have N threads, then N copies of this code will be made and they will be dispatched to each of the threads which will run on a separate core. So, we can always have less cores than N, but then a code will have to run multiple threads, in a multitask time shared kind of manner.

So, the first is get my processor id, let us say that is a get my core id. So, int myId is omp get thread num, this is an OpenMP statement, which gets the thread number; it gets the number of the thread, whether is 3 or 5 or 7, it just gets the number of the thread int myId. It adds a portion of the numbers.

So, what is my portion? Well, my portion starts from myId. So, core 0 gets 0, core 1 gets 1, core 2 gets 2 and so on. So, I find the starting and ending index of my portion of my array, which is essentially myId into SIZE by N and end index is start index plus SIZE by N.

So, all that I have is that, I have a simple for loop that simply iterates through this portion of the array and then adds up all the numbers and store them in partial sums myId. Of course, here the implicit assumption is that, the entire partial sum each of these entries in partial sums, each entry initializes to 0 that is the assumption.

So, after making that assumption, all that I do is; I add up all the numbers for my part of the array. So, each thread does that. So, considering a large array, if let us say is broken into N parts. So, thread one. So, let us come from 0 thread, 0 would add everything in this junk; thread 1 would count everything from this, 2 would count everything from this is so on and so forth. So, now, all of their partial sums was stored in the array partial sums, subsequently, the parallel section completes.

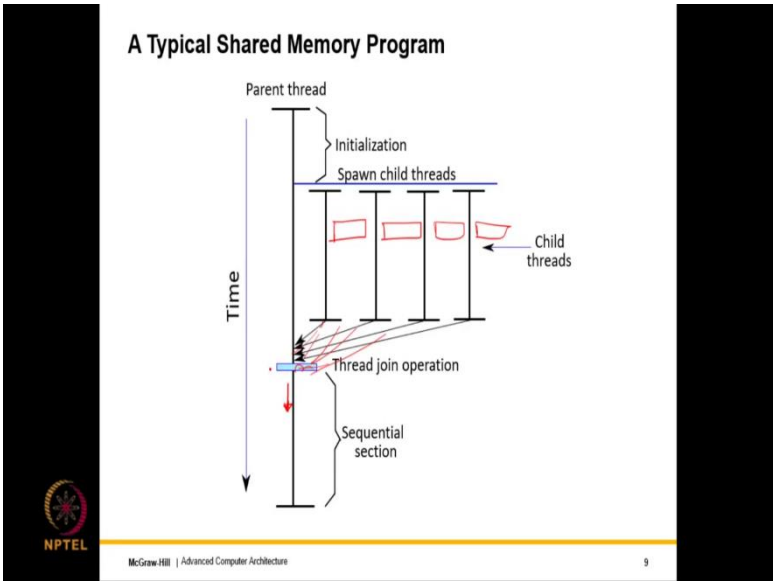
So, this was the line share of the work, where the large problem was divided into small problems. So, this part completes now, then the sequential section begins. In the sequential section all that we do is that, a single thread which would be the thread that initiated this parallel processing; so the control returns back to it, it iterates sum 0 to N and simply adds up all the partial sums and stores them in the variable result, which as you can see was initialized to 0.

So, this completes our processing. As you can see the primary feature over here is, we have a parallel section and then we have a sequential section after that, the map portion of the work is done in the parallel section. And we are using two OpenMP directives here, # pragma omp parallel to run a piece of code, copies of a piece of code on each core. And furthermore, we have a get processor id is essentially get core id, which gets the number of; say it is actually in well, I should be more accurate over here, actually get thread id.

And as I said we can have many more threads or number of cores and what we really care about are threads. So, this gets the thread number and a threads are number 0 to (N – 1). So, from the thread number, we can find out the starting and ending index of the portion of the array assign to let us say thread I. Then all that we do is, compute the partial sum for thread I; the same happens for the rest of the threats and then we add the partial sums.

So, here the simple directive is doing much more than what you think it is doing and it is handling the entire code of creating the threads, setting their state and dispatching their code to them. And the get thread now is very simple; because even if let us say N copies of the same code are made, they will still return different values for this function and the values will range from 0 to (N – 1).

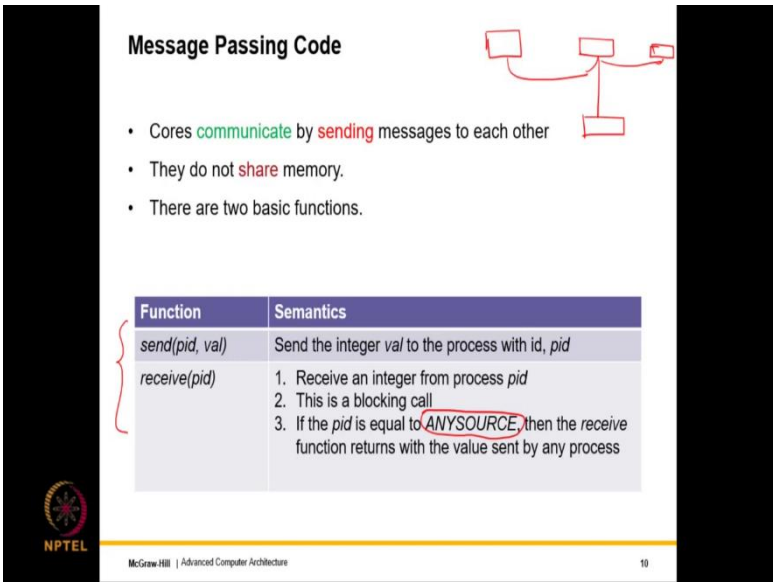
(Refer Slide Time: 19:23)



So, given that you have seen this, let us see a quick visualization of what just happened. So, we had a parent thread, we initialize the parent thread. Then we spawned a set of child threads. So, set of child threads all did the job of taking a portion of the array and adding it, portion of the array and adding all the elements.

Then they wrote their values to the partial sum, the thread join operation, where all of these threads can have terminated and join the main thread. The main thread subsequently began; it was sequential section, where all the partial sums were added and the variable result was finally computed.

(Refer Slide Time: 20:08)



Now, let us look at another paradigm. So, which is not really used in multi cores systems; it is used in multi processor systems, where the multiple processors are connected over a network. So, here the idea is that, the cores communicate by sending messages to each other.

So, we can say that maybe one processor is over here; then they are connected to the other one via internet. And so, there can be a complex network of processors, processing nodes rather and the network cards. So, the connection will happen on the basis of sending messages. So, the messages are sent to the IP addresses; the standard network based communication the way it happens on the internet that one processor will send a message to another and they can be connected with any arbitrary topology it does not matter.

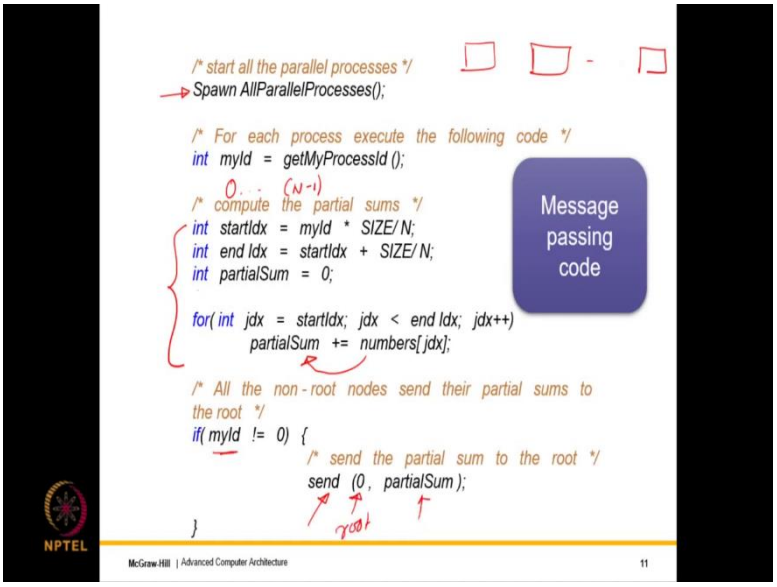
But the messages are pretty much sent using the standard networking TCP IP networking protocols. The memory is not shared. So, as I said all communication is by sending and receiving messages; there are two very basic functions that MPI, MPI is essentially a Message Passing Interface that MPI supports. So, I do not claim to be explaining MPI per say in great detail and the program that is shown, it is like MPI, but it is not MPI with the exact syntax.

So, one function can be sent pid, val, where I send an integer val to process with id pid. So, it can be process with the id pid or it can be. So, this an MPI process. So, the MPI

process will be map to some machine and some IP address, but let us for the time being forget about that.

Receive pid's, receive an integer in this case from process with id, pid; this is a blocking call which means until we do not receive, we just wait, we do not do anything else. In a pid = any source, then it means that as long as someone sends a message; even if someone sends a message, the receive function will return with the value sent by that process. So, we, do not want a message from a given process, any process will do.

(Refer Slide Time: 22:47)



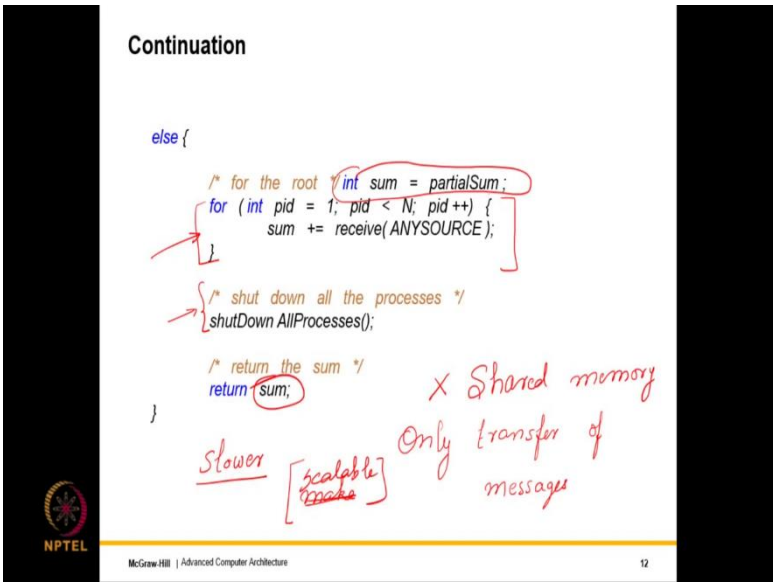
Say given that the message passing code, which looks like MPI, but it is not MPI is like this, we start all the parallel processes. So, we have this directive over here small all parallel processes. So, what this directive does is that, if let us say we have configured it. So, I am not showing that code; if you have configured N processes, then pretty much over the machines on a network, it will initiate those N processes.

For each process execute the following code, int myId = get my process id. So, we had get thread num in MPIs; in this case we get a process id, which will again return the same thing and integer between 0 and (N – 1). We compute all the partial sum. So, this code for computing the partial sums are exactly identical with one difference, we do not have an array.

So, in the other case, we used an array; because we shared memory. So, in this case we do not share the virtual or physical address space. So, what we do is that, we definitely do not share the virtual address space. The physical address space we may share; if multiple processors are running on the same machine, but that is irrelevant. So, the partial sum here is initializing to 0; we add a portion of the array and we store the sum to the local variable partial sum.

So, one of these processes we designated to be the main process or the root process. So, let that process have a process id of 0 and let the rest of the processors have an id which is between 1 and (N – 1). So, if it is a non root process, meaning the myId field is not 0; then we send the partial sum to the root. So, we invoke the send function, send it to process 0, which is the root and what do we send, we send the partial sum.

(Refer Slide Time: 24:50)



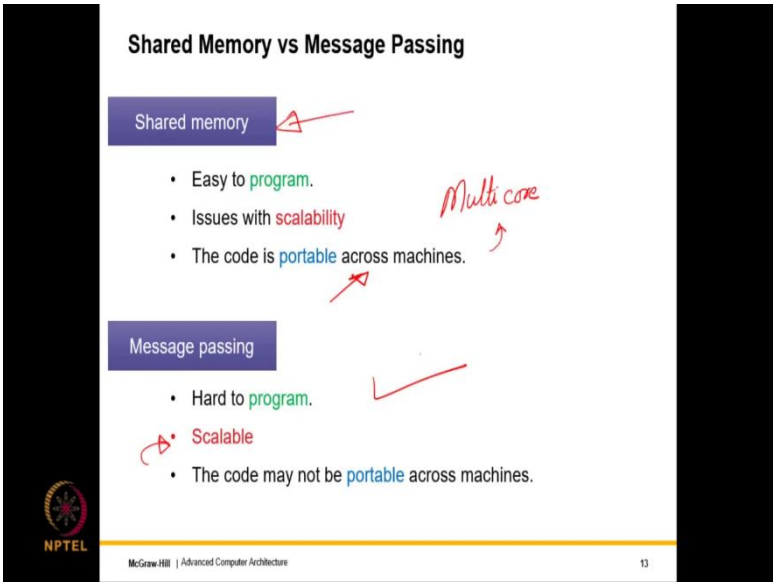
So, the map process ends here. Now, for the root, we have int sum = partial sum. So, may, so initialize the sum value with what the partial sum the root has computed. Then for the rest of the processors, we run this for loop. As long as, we receive a message from any source, we add the integer to sum. And given the fact that we know how many messages we will receive? which are (N – 1) messages, we just run this loop for (N – 1) *.

And after running it for (N – 1) *, we will get all the messages, we will add them up. Then a little bit of cleanup is required, little bit of book keeping, all the processes will be shut down. And the sum variable will be return back to the caller. So, here clearly there has

been no only transfer of messages, only messages have been transferred without having a shared virtual address space.

So, this is clearly slower, but more scalable. So, as compared to shared memory; this is undoubtedly slower, because of the overhead of sending messages. But clearly it is more scalable. So, a shared memory maybe will not scale more than 200 cores, where each core is running a thread; but message passing can scale to thousands or hundreds or thousands of nodes. So, basically this thing is you can say made to scale; it is slower, but more scalar.

(Refer Slide Time: 27:01)



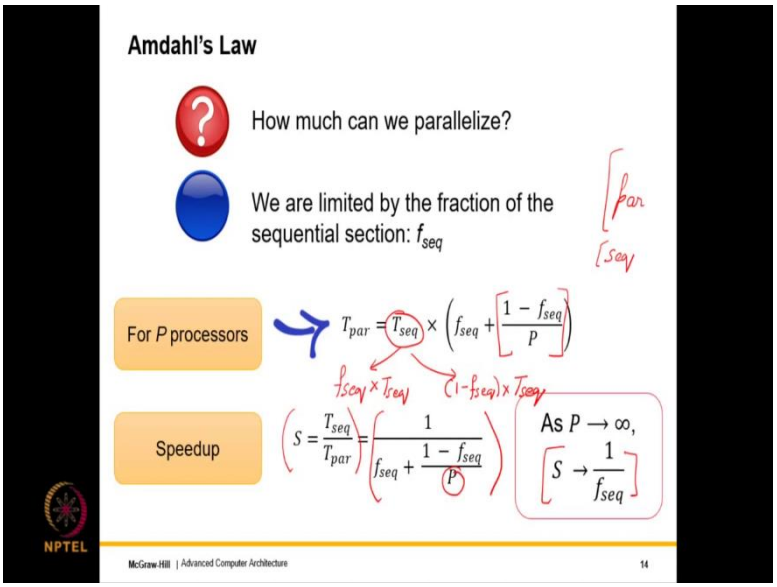
So, now let us keep going. So, the summary is shared memory of course, easy to program, not scalable and the code is portable across machines. So, across different machines, we just initialize the thread. So, threads are scheduled on the codes using a scheduler, a scheduler in American English.

And it is kind of portable code, in the sense that code that runs on one machine will run on another. MPI is also portable, because it is a standard; but nevertheless, it is hard to program, if we are relying on hardware IP addresses, then of course the code is not portable at all.

But in many cases, so the MPI interface nicely abstracts all of that; in that case, the portability is much much more and clearly scalability is there, in a sense that message passing is always more scalable. But it is not really the method of choice or what we are

interested in, which is the multicore system. For a multi processor system like a data center level super computer, message passing is great; but not for what we need, so we will primarily focus on shared memory.

(Refer Slide Time: 28:15)



So, to focus on shared memory, we will not be able to move much further, unless we do a little bit of math. So, this mathematics is embodied in the Amdahl's law. The Amdahl's law predicts that, if you increase the number of processors; what will the scalability of the system look like. So, the question is how much is it that we can parallelize? That is what the Amdahl's law predicts.

So, we are limited by the fraction of the sequential section, which is f seq. Because as we have discussed program a parallel program in particular has two sections; it has a sequential section, where a single thread runs and we have a parallel section, where the multiple threads run. So, the Amdahl's law is by and large the key mathematical determinant of how my speed up we tend to get if we have P processors and a program where we have a parallel section and a sequential section.

So, for P processors, the time that the parallel section actually requires is like this; it is not the time that the parallel section requires, it is the time that the parallel execution requires or the execution in parallel mode requires, it is a time required by the overall sequential execution.

So, the sequential execution can be broken down into two parts; one is f seq the sequential part of the execution and the other is the parallel part of the execution, which is the (1 - f seq) the baseline which is the sequential execution type.

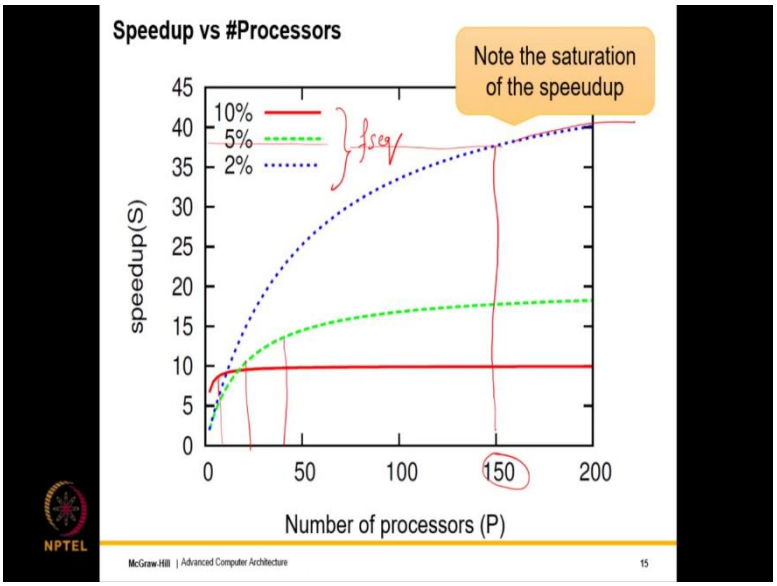
So, when I have P processors, what I am in the sense speeding up is, I am speeding up the parallel part. So, make a rather ideal, idealistic assumption over here that; if I have P processors, then I am speeding up the parallel part P times. So, this is typically not the case, we typically do not see a linear speed up; but let us assume we do as in this problem of dividing an array we actually will.

So, what we will actually see is that, the time required by the parallel section will be get will get divided by P the number of cores, the number of processing elements. So, the time the parallel time will be given by this equation. So, the speed up which is $\frac{T\ seq}{T\ par}$ will actually be given by this equation, which is $\frac{1}{fseq+\frac{1-fseq}{P}}$.

So, as I increase P, this quantity the denominator will decrease. So, the numerator remains the same and the ratio will increase. So, the speed up will increase, but here is the most important part, the crux of the argument; there as P $\rightarrow \infty$, the S $\rightarrow$ $\frac{1}{fseq}$.

So, the essentially what will happen is the speedup will saturate. It will asymptotically saturate. So, beyond a certain point for a given problem with a given sequential execution part, regardless of the number of cores that we add, the speed up will remain the same.

(Refer Slide Time: 31:54)



So, that is the key message here. So, what I am doing here is I am showing you a graph where the x axis is a number of processors. And so, I am increasing that from 1 till 200 and y axis is the speedup. And the different bars pretty much show the different the fraction of the sequential part.

Then the sequential part is 10%, then the speedup does saturate at 10 and that to quite early. So, beyond this point, increasing number of processors has no effect at all. Then I will show you for 5%, which is the green line; here also signs of saturation at this point which is roughly 25 processors, beyond that there is no point in investing the improvement will be slow and gradual.

We can may be take it up till this point which is may be like 42, 43 processors, not beyond that. In comparison with the sequential part is 2 % are ultimate speedup will be large. So, as we have seen it will 10 = $\frac{1}{f\ seq}$, which is 50. So, we can see that up till around 150 odd processors, may be 200 it is saturating, but somewhat gradually and it will ultimately saturate though, it will it is saturate somewhat gradually.

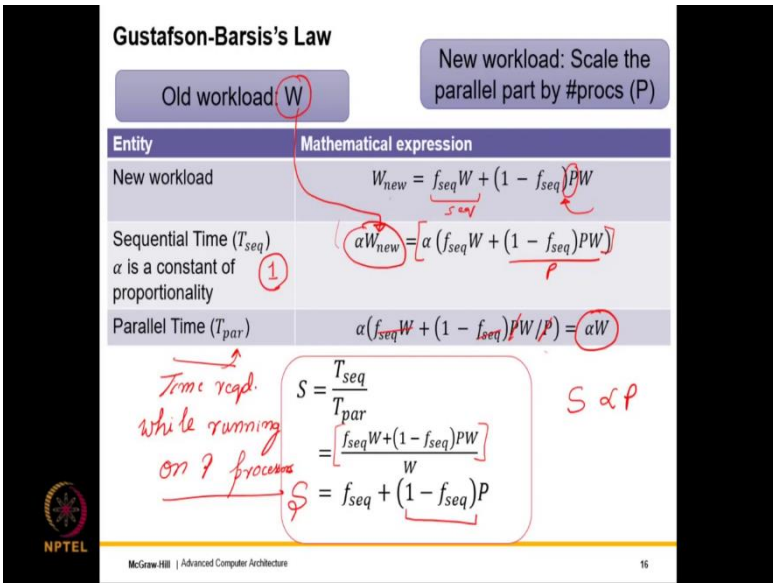
So, we can if we invest 100, the money for 150 processors we will get a good speed up, which is a roughly like this, it is roughly like 37, 38. So, the most important point is that, the key determinant of the overall speedup is the sequential execution fraction f seq.

And that determines what will be the ultimate the final speed up; even if I have infinite number of processors and it will saturate in the asymptotic manner totally determined by this equation, which means that look if the problem is such that f seq is high, there is no point in investing a lot of processors.

So, it shows us the limitation of the parallelism, limitations of the parallelizing activity and it also shows us the amount of saturation that can happen. So, when you are considering large 1000 core or 10000 core nodes, so this is by the way hold for both shared memory as well as massage passing.

So, when I am considering large supercomputers, the problems that need to run on them; they need to have extremely small sequential execution parts, because otherwise that if we do not have that, Amdahl's law will dictate that, we saturate and that too be saturate soon.

(Refer Slide Time: 34:51)



Next I will describe the Gustafson Barsis's low. So, this remedy is a crucial flaw or rather short coming of Amdahl's law. So, Amdahl's law had an assumption that the amount of work, the problem that we solve; let us say when we have 10000 cores remains the same, that will not be the case. See the old workload is W, let us measure it in terms of time, then the new workload.

So, we can measure it in terms of time, instruction does not matter; the units are not important, whether new work load if let us say we have P processors, it needs to be much

larger, it cannot be the same. If I have 10000 processors, I cannot be solving the same problem, I need to be solving a bigger problem; because let us think about it, on a smaller machine like a laptop, we solve a small problem, let us say like doing our assignments; but on a bigger super computer, we try to predict climate change.

So, we scale the parallel part by the number of processors which is P. So, the new work load, again with arbitrary units = f seq * W, which is the sequential part. this (1 - f seq * W) used to be the erstwhile parallel part; but it is being scaled with the number of multiplied with the number of processors P. So, the parallel part is being scaled by a factor of P. So, this becomes the new workload.

Say this is a new work load, let us say that this is linearly dependent on the time and the time sequential time can be, since linearly dependent will have a constant of proportionality, let it be α. So, then we can have α * W new = α times this entire expression and the parallel time over here.

So, this is the new workload, in the sense, this is a new work load for the sequential time and the parallel time or the time when we parallelize will essentially be the new workload, which is running on P parallel processors. So, similar to Amdahl's law, this factor over here right will get divided by P. So, what will pretty much happen be P and P here will cancel, f seq W and f seq W over here will cancel.

So, pretty much the parallel time will just be α * W. So, what I am doing is? So, the parallel time is the time that it time require, is the time required while running on P processors. And a sequential time is the time required while running on one processor.

Of course, we are not running the old work load, instead we are running the new work load. And the new workload is essentially the old sequential part + the old parallel part multiplied with the factor of P. So, the new workload if measure in terms of time = α * W new, which is α * this expression.

So, this is the new workload and this is essentially the time required if I have a single core, but the thing is that I do not have a single core. If I am, if I have P processors, then what I can do is, I will divide the parallel part; not the sequential part, just the parallel part, because the parallel part will be spread up. So, I will divide that by a factor of P and then you will see my terms will nicely cancel out and the time remaining will be α * W.

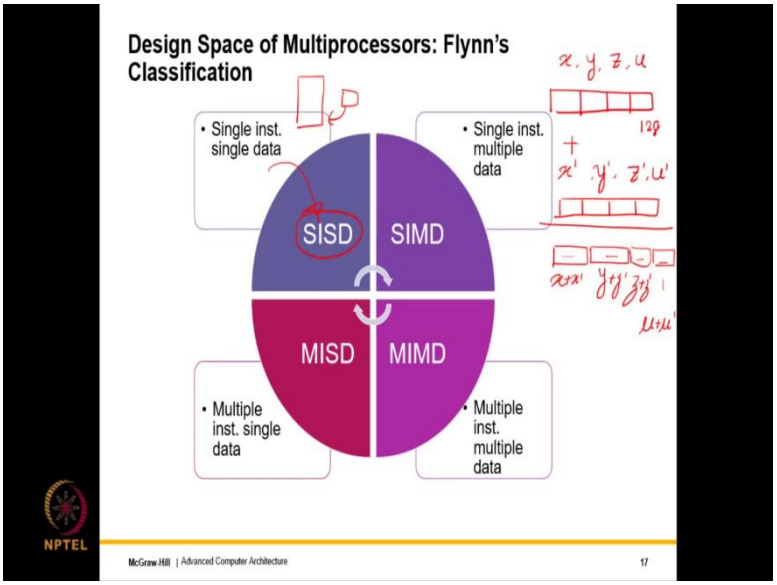
So, if I divide the sequential time with the parallel * α will cancel out; what I am left with is this expression, I will take W out. So, the final speed up with the scaled workload bind u = f seq + (1 - f seq) P, where P is the number of processors.

So, there are several interesting first over here. So, the key point is, for a large P, pretty much what you are seeing is that the small f seq over here will stop being a factor. So, the speedup will pretty much become proportional to P. So, let me see a graph after this.

But this thing can be very nicely graphed. So, what you will see is that, the speedup will become proportional to the number of processors which is a good thing and; but of course, the slope will be decided by 1 - f seq. So, f seq in this case, instead of being a limited the way it was when we were dealing with Amdahl's law; the fraction f seq over here is just deciding the slope, which is also important.

It is essentially telling us that, if let us say double the number of processors what is it that I get and that will be decided by the slope, right. But the key factor over here is that, the net speed up we will get when the workload is scaled is pretty much proportional to the number of processors. So, this is the advantage of the Gustafson Barsis's law that, it is actually Amdahl's law plus plus, in a certain sense that it is scales the workload. So, that is the most important point over here.

(Refer Slide Time: 41:30)



Let us keep going. So, let us discuss the design space of multiprocessors the Flynn's classification. So, here we will look at the entire design space of the multi processing world. So, we will start with a conventional uniprocessor. Say conventional uniprocessor, which is a single core is there in the design space. So, where we pretty much have; it is called an SISD multiprocessor, where we have a single instruction stream, a single program basically which supplies the instructions.

And a single and essentially a single data stream, a single memory which supplies the data, alright. So, we are seen that, then we will have SIMD. So, SIMD is something that was used in the good old days in vector processors. So, what SIMD is, it is a single instruction, but multiple data. So, what that basically does is that, we can let us say take four integers x, y, z and u.

So, these four integers maybe the 32 bits each. So, we will pack them into what is called a vector register, each register is 128 bits. We can have four more variables x', y', z' and u'; again we will pack them into a 128 bit vector register. So, we need to define a vector register file as well and when we add the two vector registers, we will get a final outcome. So, the final outcome; so, this entire addition adding four numbers will be treated still as a single instruction, where we will add two vector registers, the output is a vector.

So, one instruction is adding four pairs or numbers; but the result is x + x' over here, over here we will have y + y', over here we will have z + z', and over here we will have u + u'. So, we perform pairwise additions of numbers, but we perform four pairs using a single instruction. So, we have a single instruction, but multiple data's; we have multiple data x, y, z and u multiple data, but single instruction then.

Then we will consider MISD where the data is the same which means that the problem is the same, but there are different programs solving the same problem. So, this is very rare, but one very important examples are an aircraft. So, modern aircrafts are mostly computer controlled or autopilot and emergency response a lot of things are computer controlled, but computers can make mistakes, wires can get snapped, programs can have bugs.

So, what modern aircrafts actually do is that they have three separate processors made by different vendors. So, one could be Intel, one could be MIPS one could be IBM. So, then the same data which is the same sensor data is provided to all three; all three produce the

result. So, if the result is turn left by 70 degrees that is the result all three produce a result then a voting circuitry gives the final result on the basis of a majority vote.

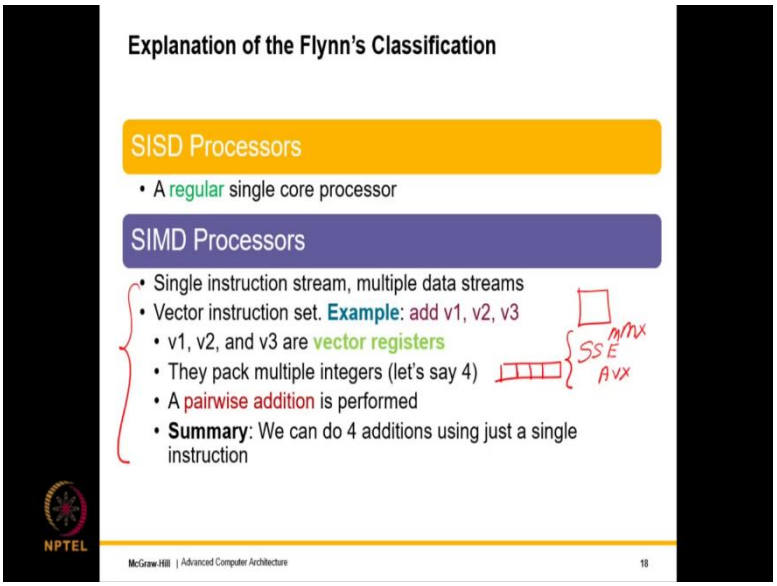
So, here what do we have in the same data single data because it all comes from sensors same set of sensors multiple instructions because these are the different programs running on different processors with different mix, but the final output is meant to be the same which should be the same most of the time not most of the time all of the time.

So, when you are running on an aircraft there is no room for error, all the time the outputs should be the same and if they are not which means if there is a fault god forbid we will go for a voting and the best two out of these three will be chosen not the best two, but the ones the two of them that match. So, we are not assuming that we have two failures at the same time.

So, the two that match they will be chosen because of the voting circuit. What we are most interested is MIMD Multiple Instructions Multiple Data and so this is what we will look at where the instruction stream is different multiple threads. The threads even if they are compiled from the same C code, but we are they are not really running in lockstep.

So, the instruction streams are different and they could very well be running different instructions as well and working on different parts of data. So, MIMD is a broad umbrella that we would be looking at.

(Refer Slide Time: 46:37)

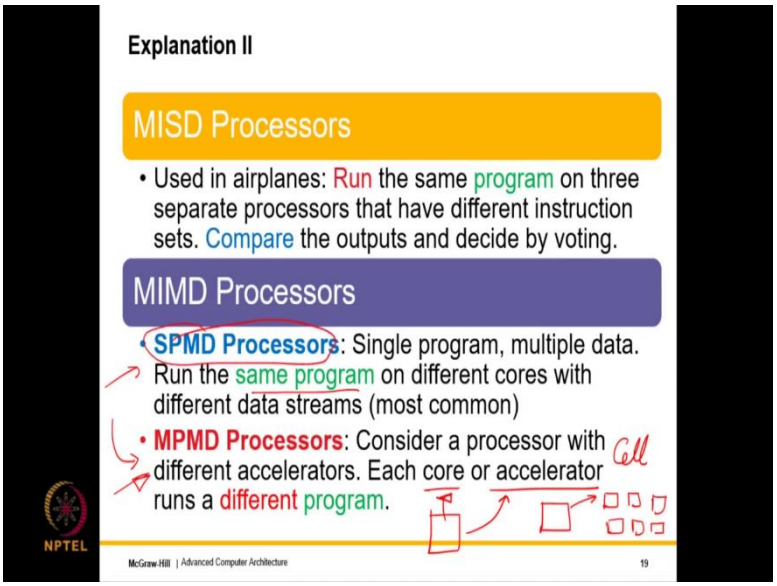


So, this is again explained. So, I would like to go over all four of these once again a SISD as I said is a regular uniprocessor a single core processor SIMD; SIMD is for a vector instruction set. So, in this case we have vector registers. So, we used to have vector processors that only had a primarily predominantly had vector registers, but most common processors such as x 86 or arm do a vector instruction to their ISA.

So, x 86 notably has the SSE 1, 2, 3 and 4. the MMX SSE and AVX instruction sets which are extensions on the base x 86 instruction set and they do provide vector capability and the vectors instructions are heavily used in numerical code and graphics code, in games and so on to speed up the most critical aspects of the computation.

And what do they do? Well they define a register which is essentially a packs set of integers or floating point values within 128 or 256 bits they perform pairwise operations. So, such SIMD in instructions are very very important in most of our code like games and other highly performance in intensive codes that require a lot of scientific and linear algebra operations.

(Refer Slide Time: 48:12)



MISD processors, airplanes MIMD, processors MIMD that in two kinds single program multiple data and multiple program multiple data. So, this is the most common where we have a single program you write the program once as we have shown in openMP. So, here the data streams vary in the sense that we work on different pieces of data and the same program is run on different cores and each core of course, runs a subset of instructions of

the program hence we say we have different instruction streams, but the program is the same. So, we do not write different programs for different cores.

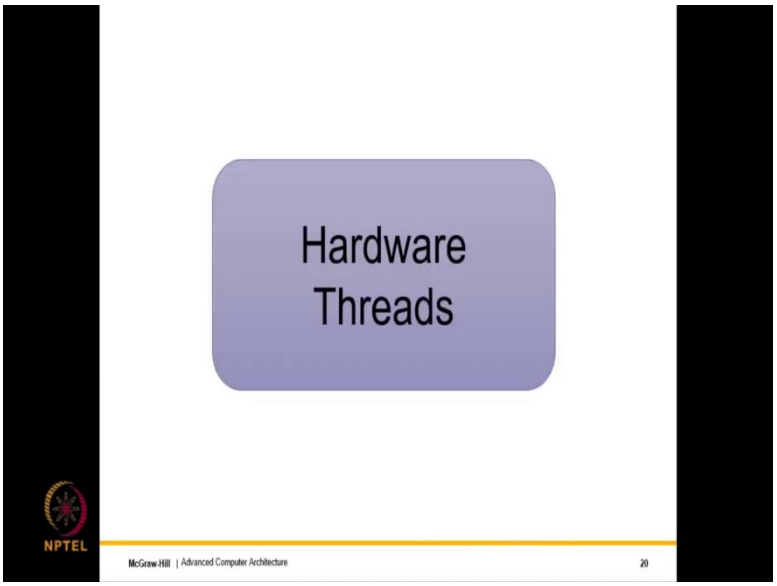
So, hence we have this get thread id and get openMP thread num primitives to essentially identify which thread it is such that the appropriate data can be fetched. Another relatively rarer cousin of the SPMD is MPMD in MPMD well what we do is that we have different kinds of processors we might have a core or a specific accelerator. So, then in the parallelism itself while programming we create a separate program for the core and a separate program for the accelerator.

So, as you saw in the chapter on GPUs any program that is written in CUDA is ends up actually being two separate programs; one program is sent to the CPU and one program is sent to the GPU and then they execute their parts. So, the programmer clearly knows which part runs on the CPU which part runs on the GPU and they are kind of written separately.

Even though they might be in the same file, but they are nevertheless written separately. So, each core or accelerator in this case runs a different program that is MPMD. We do not have many examples of this, but with modern processors that use accelerators we are started to see more of such kind of programming where we write some code for the CPU write some code for the accelerator. IBM used to have a cell processor at one point of time where there was one large big fat master code and then there were many of these small slave cores for this small numerical operations they could do them in parallel.

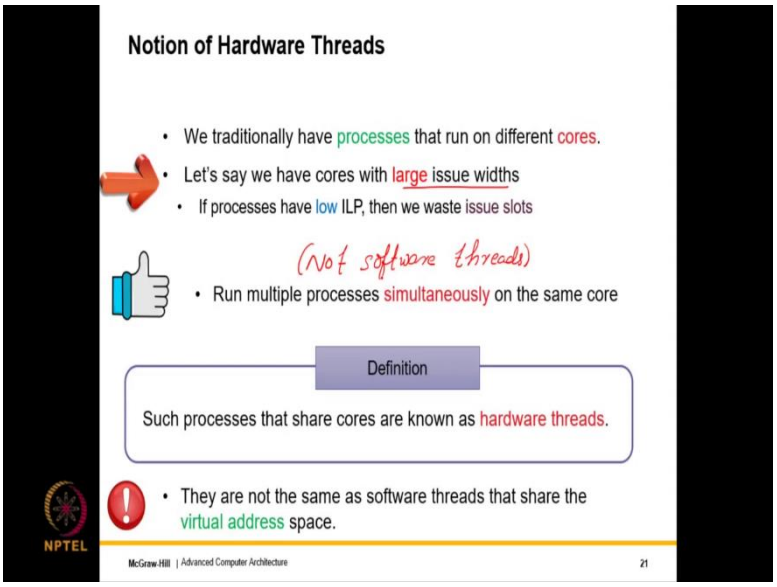
So, again the programming model was such that you would write separate core for the master core and separate cores for the smaller slave cores, but this nevertheless not that common what is more common as SPMD.

(Refer Slide Time: 50:56)



So, next we will discuss hardware threads.

(Refer Slide Time: 51:04)

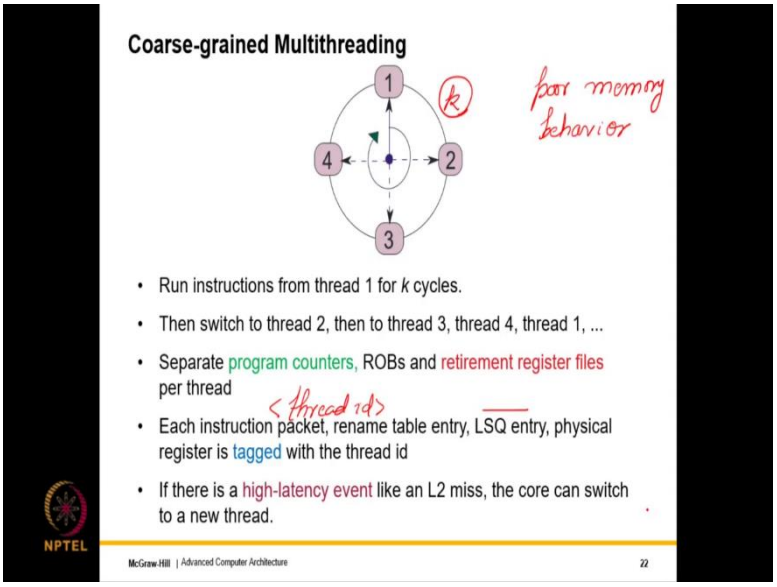


So, let us understand the basic notion of hardware threads. So, we traditionally have processes that run on different cores and what really happens is that these cores have really grown over time. So, they are cores with very large issue with large fetch width come into width. But if we have low ILP then we are essentially wasting issue slots. So, we cannot take advantage of a 6 issue or 8 issue processor if it has low ILP, the reason is if we can only sustain an ILP of one or two the rest of the issue slots are being wasted.

So, why can't we run multiple processes in this case? They are not threads that share the address space they are multiple processes simultaneously on the same core. So, the point that I would like to underscore over here that these processes do not share the address space. So, they are not software threads in the sense in which we have been talking about them.

So, let us say they are not software threads they are separate processes which do not necessarily share the address space they could I mean they could be software threads, but again that would be a specialization of the overall generic concept the generic concept is that these are multiple processes which can be unrelated to each other. But these processes share the same core in the sense in the same window of time they execute concurrently and they are known as hardware threads they are not the same as software threads they are hardware threads.

(Refer Slide Time: 53:04)



What is the advantage of having hardware threads? So, we have three kinds of hardware multithreading as it is called and in server processors all these three kinds of hardware threads are extremely common. So, they are extremely common and you will find very few server processors as of today and today is 22nd of December 2020 where you will find a server processor that does not support hardware threads to some extent.

So, I will tell you why its that important? So, what we can do? Is let us say we run instructions from thread 1 for k cycles. So, in this k cycles, what will happen? So, typically

we will encounter some long latency event in the sense that there will be a couple of memory requests those memory requests will go down to the memory system and fetch their data.

And we also need to understand one thing why is the ILP of some code low why do we see low IPC at the end? The reason we see is essentially because of memory misses and also it can also be that the dependence structure is such that we will not be able to run many instructions in parallel because it's a snake like code where there are lot of dependencies that can be the case, but most of the time it is because of poor memory behavior.

We see a low IPC because it we might have high ILP, but we will actually see a low IPC because of it. So, let us say that it is poor memory behaviour like poor cache locality. So, in that case what we can do is we can execute a few instructions we will not be able to make a lot of progress.

Because the data would not have come back then we switch to a thread to execute a few more instructions for let us say k cycles then thread 3, thread 4 and we come back by the time we come back to thread 1 our expectation is that a lot of the data from the memory system would have arrived.

So, what do we need, what do we need at the level of the core at the level of the hardware to support such kind of multithreading? So, what we need is that we need separate program counters one for each hardware thread separate robs can be the same ROB, but then well it has to be a special kind of ROB that allows instructions or thread 2 to retire even the instructions of thread 1 are before it hence better to have separate ROBs. Separate check pointing state and retirement register files.

Furthermore, rename table entries etcetera can be shared in the sense we can have one master rename table, but of course, the architectural register the instruction packets have to be tagged with the thread id. So, the thread id will tag many of these structures including LSQ entries. So, if the LSQ storing physical address then of course, we can forward between threads across threads, but if it is storing virtual addresses.

So, recall our discussion on whether LSQ store physical or virtual addresses in chapter 7, but if it is storing virtual addresses with some additional guarantees then the thread id

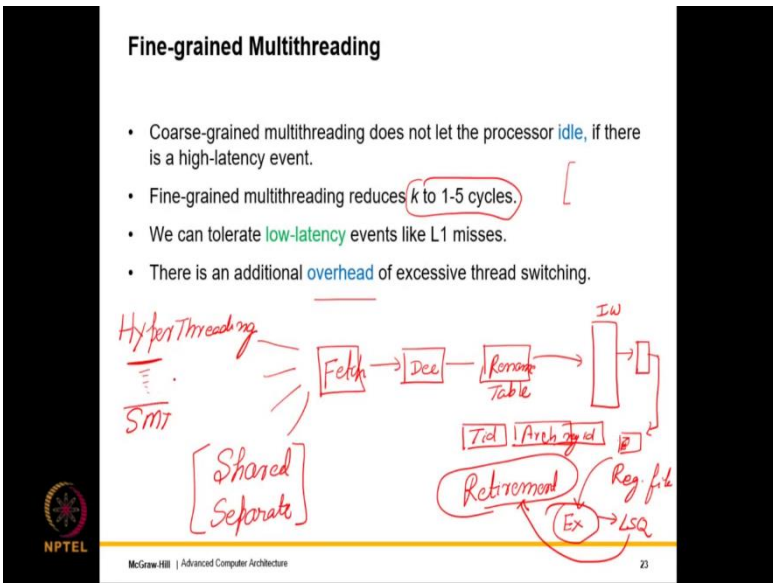
needs to be there, additionally we have some options with respect to physical registers. So, I have deliberately not mentioned that, but this is a right time to mention see either they have separate physical register spaces. So, in that case, the physical register should be tagged with the thread id or if the share the physical registers then of course, the rename table will take care if you have a single free list.

And then what it will do is, that it will just assign physical registers to the different threads as it normally does, but it of course, we will have to ensure that the partitioning of the physical registers across the threads is perfect in the sense it's never the case that we create a dependency or we do not respect a dependence between the producer and the consumer because of this.

So, this is doable as I said there are two ways either separate physical register spaces where we tag it with the thread id and so that will kind of indicate that which thread does it belong to that is one option and a next is that if there is a high latency event like an L2 miss. So, then also we should switch from one thread to the other which means we need not wait for k cycles.

Let us say in the middle there is an L2 miss. So, then at that specific point of time we can switch. So, for some of these very high latency events coarse grained multithreading is a good idea. Let us now specialize this description a little bit more and look at some of the finer points.

(Refer Slide Time: 58:41)



So, we will move to fine grained multithreading fair let us see that you know let us look at let us try to optimize this further. So, always if there is a high latency event which means that if it is let us say in 100s of cycles then coarse grained multithreading is a good idea where we reduce k to 1 to 5 cycles and we can quickly switch between the threads in the sense that we can quickly switch between instructions of thread 1, thread 2, thread 3, thread 4 something like a rapid context switch then we can even take care of events like L 1 misses.

So, of course, in the out of order processor and missed in L 1 is not a major event because we might find enough independent instructions assuming we do not and we find some of these low ILP phases then we can quickly switch to another thread and the switching interval can be made kind of low let us say between in the ballpark of 1 to 5 cycles maybe can be more slightly more though.

In that case, if you are switching between these threads then this would be called fine grained multithreading, but of course there are overheads. So, what is the fetch unit needs to do is it needs to be smart and of course, we will need to have separate branch predictors for the different threads otherwise they will end up polluting each other that is point number 1, decode unit can be shared.

So, maybe you can look at the pipeline. So, fetch unit of course, has to be separate. Decode unit can be shared because it does not maintain any state the rename table; the rename table will become more complicated because in this case the architectural register id is a combination of the thread id and the architectural register id.

So, once that is there we can proceed henceforth. So, as we have discussed for physical registers either we can share all the physical registers and simply have a free list which is also shared. So, that will not cause any issues in correctness or what we can do is we can have a separate physical register file for a separate thread and then the dispatch stage where we write to the instruction window.

So, the instruction packet would anyway have the thread id. So, it will never be the case that we will actually broadcast and wake up values it will never be the case that thread 2 will wake up a value of thread 1 and that will be ensured by essentially the rename stage whether tags are set in such a way that this will never happen and this will not happen

because the architectural register id is kind of being prefixed or suffixed with the thread id.

So, it will never be the case that will actually create a false dependence. After the instruction window will select the logic will change because we will have instructions belonging to different threads and in an out of order pipeline we might have fine grained multithreading here also where we say that for 5 cycles we will select instructions from thread 1, then 2, then 3, then 4.

And then the after select we will have the register file read. So, the register file read since we have the physical register id here again two options if you share the file then well we can just go and read it if you have separate register files the thread id is required to take us to the correct PRF and then the execute stage well different units such as the floating point unit for example, it does maintain state per executing thread.

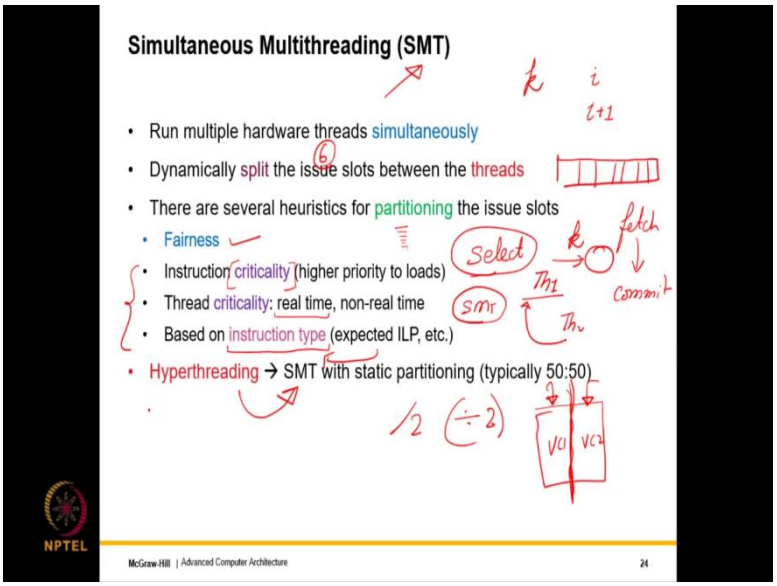
So, this state management is going to be important. for example, in the floating point architecture of x 86 the state of the unit is important or let us say we have a flags register decisions are taken on the basis of the last comparison. So, with multiple threads this is going to become slightly more complicated.

And after that when we let us say we at the LSQ again physical addresses nothing much is required, but virtual addresses we need to target with a thread id and finally, when we have retirement we need to have separate robs that is required for this purpose. So, one thing that should be rather clear is that we have a choice between having shared structures in our pipeline between the threads and having separate structures. So, there are benefits to both.

So, let me first discuss one end of the spectrum then I will discuss the other end of the spectrum in the next slide. So, one end of the spectrum is hyper threading. So, many of you who have been buying Intel processors would have seen the word hyper threading. So, hyper threading supports parallel threading to some extent.

So, we will discuss this in the next slide is one end of the spectrum and the other end of the spectrum is SMT. So, let us first discuss these terms and then we will describe the spectrum in between.

(Refer Slide Time: 64:19)



So, simultaneous multithreading as we have discussed or is the case where the hardware threads run simultaneously and what we do is instead of fetching and executing. So, instead of the coarse and fine grained approach where essentially we say that look the next k cycles totally belong to thread 1. So, then what does totally belong again is the subject to interpretation, but we can always say that look for the next k cycles will only fetch and execute an issue instructions belonging to a thread i.

The for the next k cycles only to thread (i + 1) that would be a coarse grained or fine grained way of doing things, but the SMT way of doing things would be that let us say if we have 6 issue slots and we have let us say 4 threads in the system we can dynamically split the issue slots between the threads.

So, we can say that look we have an instruction window full of instructions belonging to different threads. This cycle I will pick two instructions from thread 1, 2 from thread 3 and one each from the rest of the two threads I can say that. So, then what will happen is that pretty much all the instructions in the pipeline are executing together and that would anyway happen because coarse and fine grained multithreading in a large out of order processor would ultimately lead you to SMT because the point is that we will in the any case.

Because of these high latency events we will have instructions floating around from different threads unless of course, we are talking of something like an in order processor

with a short pipeline where for k cycles we execute instructions from thread 1 and then we flush the pipeline again bring in thread 2 again flush the pipeline again come to thread 1 and so on.

Then of course, it does make sense to totally give a processor to one thread, but in the out of order pipelines where there is a large time lag between fetch and commit with. So, many stages we will end up having instructions in any case from the different threads even if we say that look k cycles belong to you and k cycles belong to somebody else that is going to be an inefficient use of resources because some resources will remain idle.

So, it's much better to consider all the threads together at the same time holistically have some fairness criteria and have a few more criteria which we will discuss and partition the issue slots or for that matter all other time slots can be intelligently partitioned between the threads such that number 1 they are scheduled in a fair manner, 2nd some of the critical instructions which can increase the ILP at a later point of time such as load instructions get scheduled.

And also, if you have a pre existing priority of threads like a thread is real time it is processing video or something it gets higher priority and if let us say we have learnt some instruction patterns like an instruction based on instruction types for example, if we have learnt that these add instructions are very critical because the determine the result of arithmetic operations as well as they compute the values of memory addresses hence let us give them more priority.

So, based on in a such kind of thumb rules we can design a select mechanism basically. So, the select becomes rather important over here to choose which instructions ultimately get issued which means that even if a lot of instructions have woken up and they belong to multiple threads which are the ones which gets selected and then move on to execution.

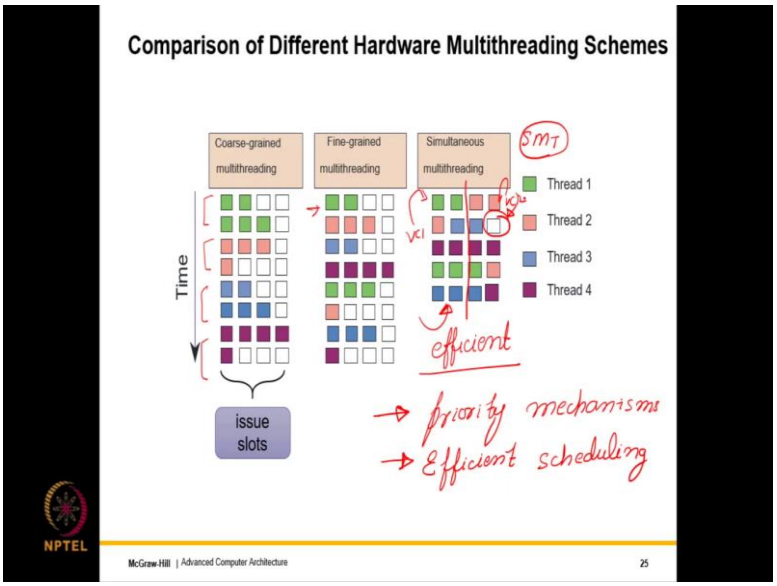
So, this entire thing is called issue and so these issue slots issue slot partitioning is the most vital aspect of an SMT processor and as I said coarse and fine grained multithreading are old ideas which make a lot of sense in an in order processor with a short pipeline or let us say when k is very large. So, let us say we are on one thread for 200 cycles and run one more for 200 cycles.

So, the overlap in terms of their instruction footprint will be low otherwise by default we would like to prefer SMT hyper threading is essentially SMT at the other end of the spectrum various static partitioning. So, all the resources are divided by a factor of 2 or maybe I should use the division symbol like this divided by a factor of 2 which means that starting from the instruction window to the functional units to the register file essentially everything the entire core is divided into two virtual cores; virtual core 1 and virtual core 2 and this includes the issue slots as well.

So, this will allow two threads to run where we have simply cut the resources off the core and partitioned it equally between the two virtual cores. So, one virtual core can execute thread 1 hardware thread 1 and the other can execute hardware thread 2. So, this is still not all that efficient and you will do lose flexibility, but let us say if you have a homogeneous workload hyper threading is sometimes very useful for increasing the throughput of the system.

So, hyper threading by a comes by default on almost all majored Intel machines, but for larger server processors such as power 8, power 9 and so on we do have SMT as the metal of choice.

(Refer Slide Time: 70:30)



So, let us now graphically compare them. So, let us have 4 threads as you can see we have threads with 4 separate colors' green, pinkish, bluish and purplish. So, then we have coarse

grained multithreading where as you can see for two cycles we issue only instructions from thread 1, then thread 2, thread 3 and thread 4.

So, you can see the degree of idleness over here then even if we cored issue more instructions we do not. So, when we are saying coarse grained and fine grained is basically at the level of issue it's not that much at the level of fetch we can still have parallel fetch, but it is essentially at the level of the issue; the issuance is where we do the partitioning.

So, that is a traditional connotation. In fine grained of course, we switch every cycle. So, we so, but as we have used different numbers in the slides in the previous slides. So, there it was written that coarse grained multithreading for 10 cycles you are on one thread and for 10 one more, but that would be hard to visualize.

So, let us visualize it kind of in a simple fashion. So, here every cycle there is a change, but here also what you see is there is a degree of wastage there is no doubt certain degree of wastage of issue slots which is not desirable. So, what we actually do is that we take simultaneous multithreading where we have a smart scheduler of instructions so to speak that nicely packs all the instructions.

So, what you can see over here that it nicely packs all the instructions. So, we have 5 green instructions here see 5 green and 5 of these purple ones and then 5 of them and similarly blue 5 of them; 5 of them. So, the accounts as you can see are the same, but they just packed more efficient. So, we just have a little bit of wastage over here, but other than that we do not have any wastage of issue slots. So, this is far more efficient in terms of throughput for sure.

Write in an in a multithreaded environment you can clearly see the higher throughput and the higher utilization of the issue slots. So, then one question can be that why is it not that common well the point is that there are overheads we need to create a scheduler which can actually do this identify and do it at runtime plus we have different instruction criticality metrics in the sense there might be some instructions on which many many more instructions are dependent.

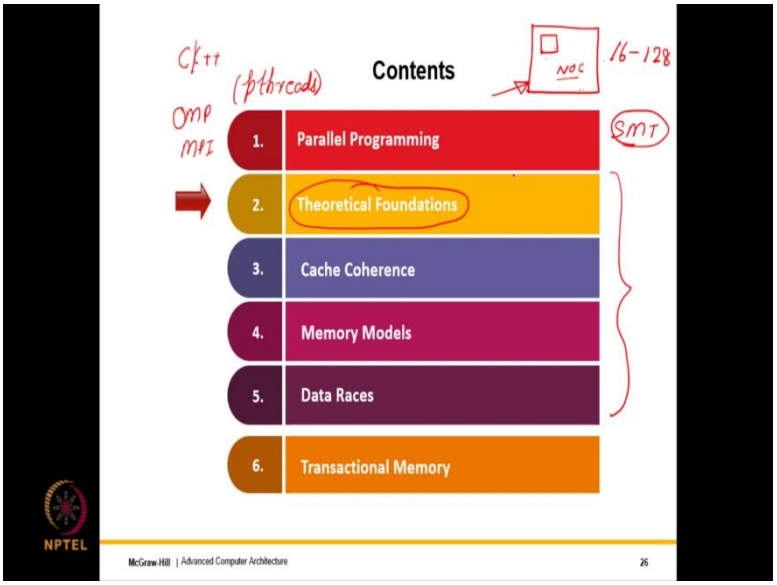
So, then those instructions should be given higher priorities a lot of priority mechanisms basically are there if we want to do this rather efficiently. So, this used to be an extremely hot area of research a while ago a couple of years ago. So, even now also there are a lot of

research peoples coming out, but it's kind of saturated, but still the key core most important idea is that different threads have different priorities can have and different instructions within threads can have different priorities.

So, efficient scheduling such that we are maximize the throughput and also provide some part thread quality guarantees this is a difficult problem there is a lot of work in this area. So, lot of these problems have also been solved to a very large extent. So, SMT is clearly the most flexible, but if I would take Intel processors they would consider only two threads and then they were hyper thread.

So, hyper thread basically means that you draw a vertical split this part belongs to virtual core 1 and this part belongs to virtual core 2, when we have two threads that is.

(Refer Slide Time: 74:47)



So, we have completed most of parallel programming at least the simple parts. So, we discussed openMP; openMP is the language of choice for writing parallel code on most machines, it's nice, easy and simple people also use the pthreads API which is at a lower level as compared to openMP, but both of these are essentially augmentations of C C + + additional libraries over C C + +.

So, in this case, pthreads is no doubt more flexible, but openMP is more popular if our message passing systems we use MPI. Then we discuss the Flynn's classification of SISD, SIMD, MIMD, MISD processors and finally. So, then along with that performance aspects

were discussed Amdahl's law Gustafson's law which plays a bound on the speed up and finally, we discuss a little bit of hardware threading where a hardware thread is definitely not the same as the software thread, but it is just multiple programme utilizing the resources of a core which is over designed at the same time.

So, what this does is that if you try to overlap the low ILP phase of one thread with the high ILP phase of another thread such that the resources of the entire core can be used the most optimally and this is more useful for server programme where you have a bunch of threads bunch of processes and we want all of them to execute. So, we are essentially looking at bulk throughput.

So, that is why SMT is the main technology over here. So, regardless of what we are using. So, modern server has a lot of technologies and is it clearly has a huge cache and it has a large number of cores. So, you can expect anywhere between 16 to 128 cores. So, they will of course, vary in their strength, but nevertheless and all the cores also may not be of the same type and moreover each core can actually be running multiple hardware threads.

So, as far as we are concerned if multiple hardware 4 hardware threads are running on a core then pretty much these are like 4 virtual cores on top of this. So, the server actually is a very very complicated world with an array of cash banks and a network and on chip network to take, messages from cash bank one or a core 2 a cash bank.

There is a huge amount of things happening inside and also as we have seen hardware threads within a core. So, it's a very very complicated environment inside the package of a processor chip. Now in this can we write parallel code correctly it will turn out that we cannot unless you understand the theoretical foundations of how to do it correctly. So, this will be the topic of our next lecture which is the theoretical foundations of parallel programming in parallel systems.