PROFESSOR: As we come close to testing, we have shrinkage here. People probably left home. Hopefully, everybody who left home finished their report. So you guys have all looked into how to do the final project. And have all the ideas how to go and optimize. How many people have downloaded, compiled, and ran, and you know what's going on? OK. Good. Good. Good. Exactly.

It's happening right now. OK. Good. So I will repeat this what I said last time in here. We're going to have a design review with your masters. So just look for us to send you the information. That means when you come back from Thanksgiving, schedule it early. So they can help if you have any changes in design process.

And then we have a competition on December 9 in class here, trying to figure out who has the fastest ray tracer created. And in fact, this year there is Akamai prize for the winning team, including they have a kind of celebration and demonstration in their headquarters. You get to go get a tour with their knock and stuff like. Plus, every winning member is going to get a iPod Nano. So there's a lot more motivation now to get the fastest running ray tracer OK.

So with that, let's switch gears a little bit. So today, I'm going to talk about distributed systems. Until now what we looked at was, OK, given a box how to get something running as fast as possible inside that box. And today we're going to look at going outside the box. Basically, we want to scale up to clusters of machines.

That means the room can have 10, 15 machines. In fact, for your class, you guys are using-- how many machines do we have? 16 machines. So you are doing independently. But you can use as one gigantic machine if you can and run something.

And data center scale. This is kind of people like Google, and Amazon, has these kinds of things. And finally, Planet Scale. If you want to run something even bigger, larger. What you have to deal with, and what kind of issues you have to deal with. It's time to reboot my machine. And I have to be pressing this button probably four or five times during the day.

So Cluster Scale. So you want to run a program on multiple machines. And OK, Let me put it there. Why the heck do you want to do that? What's the advantages of-- instead of running on one nice machine, running on a cluster of machines? What do you get?

AUDIENCE: It's cheaper.

PROFESSOR: It's cheaper. That's a good one. It's cheaper to get a bunch of small machines than to buy a humongo mainframe type machine. Yes, that's a very good answer. What else?

AUDIENCE: It's very slow. It's slower.

PROFESSOR: So you would run something because it's slower?

AUDIENCE: But it is a trade-off.

PROFESSOR: Yes, so there's some trade off between speed. But it might not be that much. Even when you get a gigantic machine, there are bottlenecks in it. In a cluster kind of thing, you can avoid the bottlenecks. But hopefully, you're trying to do it to get some performance in scaling to large number of users and what not. So basically, what you want to get is-- so get more parallelism. Because now we have more machines, more calls. And hopefully get higher throughput. Definitely, because you are doing it.

Hopefully, it's a little bit of lower latency too, because if you have one gigantic system, if everything has to go through bottlenecks, it might be slower than basically having different system. So assume, just an example, if you are something like Verizon or Netflix trying to serve your videos. It makes much more sense to have a bunch of clusters of machines each doing a lot of independent work than trying to send all the videos to one machine.

Another interesting fact is robustness. So until now you guys didn't care about robustness, because something went wrong, the entire thing collapsed. There's no half baked machine. The machine crashed, your program crashed. Everything died. So you just have this fatalistic attitude. OK. It crashed. Everything is dead. So why bother?

But in these clusters, if you have a lot of machines, if one machine dies, there's many others to pick up. So you can have a system that probably has availability much higher than what you can get on a single machine.

And finally, cost savings. Because it's cheaper to do this, have a bunch of small machines. And businesses like Google has really taken advantage of that.

So there are issues we have to deal with in order to program this damn thing. And if you want to get performance, you have to program in a way to get good performance. You don't run much slower and load less performance than one box. You'll get performance and also performance scalability.

That means if you get 10 machines, you want to get some performance as if you have 20. Hopefully, you want to get a lot more performance than 20. So how do we keep things scaling in there? And also the thing's robustness. So the idea there is if you have one machine, you're fatalistic. If the machine goes, everything goes. You don't care.

But if you have a lot more machines, you want to make sure that application runs even if the machine's fails. Worse, if you have a lot of machines, there's a lot more chance of failure. So if one goes down, everything crashes still. Then your application will be a lot less robust even than a single machine, because there too many moving parts to go wrong. So you want to actually deal with this robustness. So that adds an entire new dimension in there. We are not going to go too much deeper into robustness. But that is one big thing that you have to really worry about when you go to distributed systems.

OK. What's a distributed system? So this is what we have been working so far? Can we see if we can reduce the lights? I guess up there you can't-- OK. We don't go fully dark, we'll see. Oh, that's you guys. Don't go to sleep even though light is-- there. So this should be over there and I don't have any way to darken this side.

So these are the machines we have been thinking about. We have a memory system. And more than just having a shared memory, we have cache coherence. So that means if two people want to communicate to write to this single memory location, and the lot of that data appears lower on all the different cores. So we can use that information to basically communicate to the processor. That's really nice.

So a distributed memory machine has no shared memory. So each memory is-- Now, how are you going to communicate? Message? Yeah, this is not software. You actually need something additional. Something like a network, or Internet, something behind sitting out that actually let you communicate between each other. So if you just really look at the kind of cost, this is a back of the envelope type calculation. Register is probably one cycle. Cache is about 10 cycles. If you go to DRAM, you can get about 1,000 cycles. Remote memory, going somewhere across, is, again, another order of magnitude from that.

So of course, you keep adding. And that's probably the reason that sometimes things can be slow. Because now, we have another layer that's even slower. So we have to think about it, worry about it when you're writing code for these types of machines.

So in shared memory machines, we learn in languages like Cilk. It's very nice to communicate because we synchronize via locks. And all communication via memory. Because when you write something, if you look at that memory location, everybody else will see it. And if you put the right synchronization, hopefully you will get the value you want.

In distributed memory machines, there's nothing like that. So what we see is we explicitly sends some data across. So you have what we call messages. And that means if you want to send something to-- if another person needs to look at something, we have to send it to that person.

So you have to originate yourself. Saying, I'm sending. That other person has to receive it. And they have to put it wherever you want. So everybody's address space is separate. And if you want to synchronize, you would also do it through the message. So you send a message, then the other person wait for the message to come.

And so this shows you what normally happens in messages. In the shared memory, there's nothing called message size. You write a cache line. The cache line moves. And you can't keep changing the cache line size. Hopefully, prefetcher will be good and do something nice. But you don't have that much choice.

In messages, you can compose any size of message you want. So what this graph shows is the minimum cost and average cost of different size messages. So there's a couple of things to get out of this graph. One is that if the message is even 0 length, or very small, you still have overhead. You're going to send the darn message. So if even you send nothing, it cost you some amount.

And the second thing is as the message gets bigger and bigger, the cost keeps increasing, because now you're sending more and more data. So if you really amortize the overhead cost, you are to send large messages in there. Another thing this chart shows is that as messages become bigger, the kind of the distribution of overhead is all over the map. Because now we are sending large things, a lot of other craziness happens to these things. So sometimes it can go fast, sometimes it can be pretty slow. I will get why it might be sometimes this kind of distribution shortly.

So the main point is that, that you don't send smaller messages if you can, because the overhead is too high. So why is this? Why is sending messages complicated? Till now, there's nobody sitting between you and hardware. Once you send the program run, you own the entire hardware, and after figuring out all the weirdness that's on x86 there's nothing in between you. You probably won't look at the compile code if you look at what assembly is generated you have full view what's going on in here.

Unfortunately, message passing, a lot of other things come into play. So if you want to send a message and the applications says, aha, I'm sending a message. And normally, it will do a system call to operating system. And normally, this message will get copied into the operating system. It's copying here. This operating system called the operating system wakes up. This might be when this scheduled, there's a lot of things going on.

And then the operating system has to send to the network interface card. And the network will say, OK, I can't send long messages. I'm going to break into a bunch of small messages. And put some hardware here. And it will end up in the other side. In a bunch of fragmented small pieces that the network interface unit has to reassemble into one message and deliver up. And this will probably-- it will copy back into the application.

So what that means is that a lot of other things getting involved, each optimize separately, doing a lot of different things. And so that is why you have this big unpredictable mess happening in message passing. And so there you not only have to worry about your code. You have to worry about what the operating system is doing. You have to worry about what the network is doing. You have to worry about your network card's doing. So there's a lot of moving parts in this. If you want to get really, really good performance, people have to worry about all these things in here.

So let's look at how a message works. So I hope you can see these diagrams. Can you see these? Barely? So let me say-- So I have a sending process and a receiving process. Oh, don't reboot please.

And so what happens if-- this is a message-- if we are sending without any buffering of a message, that means I am not copying it anywhere, so assume I want to send a message. I said I have a message to send. And then what happens in this model is, OK, until the receiver is ready, you have to wait. Because there is no place to send the message. So finally, when the other side says I want to receive something, it will tell this thing it's OK to send. And it will copy the data. And then after copying the data, both parts can continue.

So this is what happens if sender wants to send early. If you're very lucky, the minute you try to send, the receiver says I want it. And we have very little delay. And everything gets copied. And that's in your lucky case. In other cases, the receiver wants some data. But the sender is not ready, so your receiver has to wait until the sender wants to send it. And when this message [? is ?] [? sent ?], you copy the data in here. So this is a very naive simple way.

What can we eliminate out of this? How can we make it a little bit faster?

AUDIENCE: Buffer.

PROFESSOR: Yeah. If you buffer, what will eliminate? What will go away? Out of-- we have this overhead, this overhead, and this overhead. Which overheads can get eliminated? Wait for send can go ahead. So what happens is-- So here actually what they're showing is buffering also with some hardware support. That means I am trying to send something. And the minute I copied it out there, I can keep working in there. And somewhere in the background where you send the data, it will arrive here. And if it wants it, the data is there.

Of course, if the receiver comes early and asks for data, you can't do that. Still you have to wait, because the data is not there. However, if there's no hardware support, both has to probably wait a little bit, because you have to get the data copied. So if you have a lot of hardware support, you don't see this copy time. But if there's no hardware support, you see some copy time going in here.

So what's the advantage of this versus-- OK, tell me one advantage of this method versus this method. So of course, this one there is a lot of wait time and stuff like that. We know that. But is there any advantage of doing this one, this waiting until sending and sending it there versus this kind of a nice sending it in the background.

AUDIENCE: They're synchronized.

PROFESSOR: Hmm?

AUDIENCE: Sychronized.

PROFESSOR: Synchronized is one advantage. What else might happen? So what else are you going to do to get this kind of thing working? So in order for this to make progress, what do you have to do to data?

It has to copy. So it has to get multiple copies. So from application space. It has to get copied to operating system space. It has to get copied into the networking stack. So data keep getting copying, and copying, and copying in there.

And in here you basically don't copy. You just say, OK, wait. I'll keep the data and when you're ready, I will send it directly in here. And you can directly probably even send it to the network. And send it.

So if you're sending a lot of data, copy my old value. So this might even be better if you're sending a huge amount of data. So that's one advantage of having system like that. And of course, hardware-- if there's no hardware support, basically still you have to do some copying in here.

So this is-- what am I showing here? So what we are showing in here is non-blocking. So one way to look at that is when you're sending, when you request for send, what you can say is, OK, I continue but I haven't copied the data. I have my data in here, but I'm doing that. But what I must tell you, OK, look, this data still hasn't moved out of my space yet. So I have to worry. I can't rewrite the data. And at some point, when you say I want the data, it will go there and bring the data for you. And catch you like that.

So between this time since I don't want to make too many copies, I have to make sure that I don't touch that data. Or I have to copy it. So that's my request in here. And of course, if you have no hardware support, you have to put some time into actually doing the copying.

So this is nice. But we want to have a little bit of high level support to do this. So this is not as nice as things like Cilk, because you don't have to worry about a lot of other interesting things going on. So what people have developed is called MPI language, Message Passing Interface language. It is kind of a bit foggy. But that's the best people have these days. A machine independent way of when have the distributed systems to communicate with each other.

So--

[PHONE RINGING]

Whoops. That's not good. My phone. Sorry about that.

So what happens is each machine has its own processor, it's own memory. So there's no shared memory on a thing like that. Its own thread of control is run. And each process communicates via messages. And there is send as is needed. And that means but you can't send like pointers, because there's no notion of pointers. You actually have a data structure that's self-contained center of the site.

So here's a small program. I'm going to walk through that. So I have main. And I'm setting a bunch of these variables. For now, those are not that important. But for completeness, I have that. And then, of course, if use something like MPI, there's a bunch of setup things that you have. And so basically like cut and paste with what people normally do as we set up.

And then I have this piece of code. This piece of code, what it does is, this same program runs on multiple different machines. So everyone has the same program. But then at some point, I want to know in my machine what to do. So what I do is I check who am I? Am I machine zero? If I'm machine zero, do this. If I'm machine one, do this. So by doing that, I can cite a piece of code that everybody runs. And everybody figures out who they are. And if they are the given thing, what to do.

So here what it says is, OK, if I'm machine zero, my source and destination is machine one. If I'm machine one, my source and destination is machine zero. So I'm trying to communicate between each other.

So if you look at what happens is, first, I am sending basically to this machine. So I'm sending something into this machine, so the syntax-- I'm not going to go through that. You don't have to know that. But what you need to know is that I'm trying to send something. I tell explicitly who to send. And there has to be matching receiving that data.

Otherwise, sends go somewhere. And just it goes bad. Send here, you can send it. It can probably go bad. But receive you have to have somebody who sends for that. So the receive basically has to have matching. And then you send it that direction. And then what I do is I receive in here. And this gets sent to me in here.

So question I did send receive here. What would happen if I did also send receive here? If I reorganized these two, what would happen? If I used the same piece of code, that two pieces of code. Then I don't even have do a bit to make this two separate code. I can basically factor this out down here. I do a send, receive; send, receive here. And then send the just IDs. What happen?

AUDIENCE: It works without a buffer.

PROFESSOR: These things are what you called blocking sends. If you have blocking sends, it means that until the receiver receives it might be blocked if you are doing a blocking send. OK. That means if two guys are trying to send, nobody is receiving. You have what?

AUDIENCE: Deadlock.

PROFESSOR: You have deadlock. So that's why I actually had to do this. This is called blocking send. So instead of blocking sends-- So of course, those are finalized things and do that up there.

I can do this one. What this says is-- This is actually a little more complicated. What I'm doing is I have a bunch of buffers here. I have how many processors? I have bunch of buffers in here. I have, I guess, my ID number of processors-- no, numtask number of processors. What I am sending, say I'm sending a circular buffer. I'm sending around to everybody. Both directions. So I am sending the previous and next. So assume something is sitting in numtasks. I am sending back and forth.

So here what I am doing is basically non-blocking sends and receives. So first time issuing a receive. So even if I receive a receive, it says I have intent to receive. But I am not receiving something. I am not waiting. So I can continue. And then I am doing the same. So otherwise, if I just do just receive and send, if you do blocking is going to be deadlocked. But here I do that.

And then in this wait for all. What it says is, OK, now I issued a receive. Now wait until that receive is done. So before I use the data, I have to wait for it in there. And also, when I do the send, I am wait for all in here. So why do you think it might be advantageous to do a non-blocking receives and non-blocking sends?

So sends, it makes perfect sense, because once I have sent, I won't do anything because I don't have to wait for anything. I am done. So blocking sends is not that useful. But receives, why do you want to do non-blocking receives, then a blocking receive? Because then you won't receive. You have to wait till the data comes to do anything. Because that's what non-blocking receives means. I have to receive early and then wait for the receives to happen at this point.

What might be an advantage of doing a non-blocking receive? Anybody can think of an advantage?

It's harder. Because now you have to remove the receives from the synchronization point instead of writing one receive.

AUDIENCE: Because when sends come from other machines, we can receive it.

PROFESSOR: That might be one interesting thing because you are expecting multiple receives. You don't know what's coming first. If you do non-blocking receive, then you can be- - opt out the first guy then basically work on. That's a very good point. OK. What else? What other advantages you might have? Having a non-blocking receive?

AUDIENCE: If the receive fails, we can just have them resend it again.

PROFESSOR: Receive fails? OK. Receive fails. See, that's complicated. But another interesting thing might be space. Because when I see the non-blocking receive, I know where the data has to be. So I already allocated a buffer for that. So, if the data comes now, I can directly copy into my local buffer if there's already a received issued, if there's already a space allocated. Normally, other way around, until you see the issue the received, you don't know where the data has to be, so it has to get copied at that point. So here, you can keep the buffer, and hopefully, if you are lucky, the same hasn't happened yet, so you should set up the buffer, and then, when the data comes, say, aha, here is the matching receive. Directly put it there by passing it and copying in the middle. So that's the advantage here.

So, I am here. I did a wait for the receives here. Did the work that uses the data. And wait for sends here, afterwards. OK. Could I have moved wait for sends before the work? What happens if I have moved wait for sends before the work? Is it incorrect? How many people think this is incorrect? Is it incorrect to move this wait for sends? All the sends before the work, to move this item about? Because work is where all the work happens, I assume, that uses this data. So wait for sends about what happens.

AUDIENCE: Well, if that's incorrect, then you lose...

PROFESSOR: Yeah, you lose-- you're waiting for something that you don't have to wait. Of course, you can move these down, because that means you might start using, and try to use data that's not there. So, this has to be here. And this, basically, for performance purposes, has to be after. So, of course you have to worry about a lot of correctness issues.

One is deadlocks. So, there are two types of deadlocks. That's blocking sends and receives, what we talked about. But there's also other types of deadlocks that happen because of resources. So, let me get to that in the next slide. And the other interesting thing that can happen is stale data. In your shared memory machine, need to update the data. You know everybody's going to see that. Because the hardware takes care of that. But, in a message passing machine, it's up to you get the latest data when it's needed. So, if you don't have the data, you think, aha, I have the data, but it might not be the right value because you haven't gotten something new. So, it's up to you to basically send the data out and that.

And, also, robustness is a big issue because the fact that you have multiple machines means you can make it robust, but the other flip side is up to you to make it robust. So that means you have to figure out if a machine fails, how to respond to that. So, if you're waiting for a machine there for it fails. OK? There are a lot of issues, it time out, and then you have to go and deal with that. So, that can make the programming a lot more complicated. And if you just don't do that, your overall program would be a lot less robust than a single machine because there could be a lot more failures in the large system.

So, here's a kind of deadlock that can happen. What I am doing is processor zero is sending and processor one is sending data to each other. It doesn't have a read write deadlock because I am for sending, sending and then I am receiving, receiving. The sends here, and the receives from here, the sends here, and then receive. So normally it looks like I'm sending two things, It should go, and I am receiving that. But, assuming that I am sending huge amount of data.

OK. So I start sending, and there's not enough room for received. Just say, OK, I don't have any room to receive, I have to wait until the data, at least a data start getting consumed to start receiving. OK. If you keep sending multiple of send outs, you might do a multiple of send outs in more than one, multiple of sends here, I might get deadlocked, I might get blocked because they can have receive, receive, and he's also trying to send multiple things, I might get blocked in here. So I might get into this criss-cross situation. If you were trying to send something, but other guy can't proceed, up to get received. So even though, if program look like there's a nice matching send and receive, there's no cycles. There's a cycle created by the resource usage in here. So, if doing lot of sends before a lot receives, and vice versa, you have to be careful. If you do too much of that, it's nice to block things, move things up. But if you have too many things, then you might end up in deadlock situation. Even though, traditionally, it might not happen.

So, you have host of other performance issues that you'll deal with. So let me address couple of them and what it might shows up. So one big thing is occupancy cost. Because, when you do shared memory, minute you basically showing instructions. Instruction goes executes, you are done with and that operation is finished. When you are doing a message passing, each message is very expensive. It has to do a lot of things. You have to do a context switch, do a buffer copy, and a protocol stack processing, and then might have to do another context switch for that. And there's a lot of these copying and stuff happening, and the network controller might interrupt the private system, because either there's data coming, copy the data in there, and you send as ignored application.

So there's this huge amount of things happening just for one message. If you are sending like one lousy byte, or one even like kilobyte., it just doing a lot of millions of instructions, just on behalf of small amount of data. So that's a large amount of cost associated with that overhead. So, setup already is very high. And, so what you want to do is you want to amortize the cost by sending large messages. So what you were to say, so look I'm not sending this small thing, I'm going to accumulate a lot of things, I'm going to send everything as bulk if you can. And then you can basically amortize these costs of doing things. Other thing is communications is excruciatingly slow. So even the memory system, it's about probably a couple of hundred plus compared to CPU communicating. In the cluster interconnect, you can do tens of thousands of instructions in the CPU, by the time it get communicated. In a grid, or if you are doing through the internet, and then it sits in the seconds now. You can actually feel it. And then your processor can run millions of instructions in that time. And so if you are waiting for something, you had to wait for a very long time. They might not have enough things to stuff in the middle, to kind of amortize the cost. And then you have to worry about that.

So what that means is if you start waiting for something to happen, you are waiting for a long time. So you have to figure out putting things in there. Not waiting this, non-blocking things is very important because of that. And so normally what you want to do is you want to have always split operations, that means you want to kind of initiate something at some point, very early on, and then use it later, especially if you're looking for something like a receive. So if, I want to, especially if I want to get some-- normally in the shared memory, just kind of doing a simple thing, asking something and get replies, very simple. Here, because if you just do that, there's a huge waiting bit rate, so you want to kind of do speed operations. So, this code can be very complicated, because you tried to-- if you want to ask something, you ask very early, you do a lot of other things before the reply comes.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Oops. OK. Let's see how many times I had to press that. Before I realize I have to reboot. So, if you want to rendezvous with-- normally, what have that means that two points has to kind of synchronize at the same point. So what you want to do is you can do a three-way sender send a request, receiver acks with, OK, it's OK to send, and the senders delivers the data. So that means I have to do a three-way communication. Or, this alternative with two-way, you sender doesn't send anything, receiver basically send a request, and then you send the data. So this could be faster because there's less things to do in here.

There's another method called RMA, or it's another name you might see, it's called active messages. Where you don't ask the receiver. When you send, you have some pre-assigned place you can go and dump the data. So you don't wait for somebody to ask for data. You said, OK, if I want send, I'll immediately send, and put it somewhere. And the data that I send the place in here. So, the first slide I showed you, all at that time, you saw all these big difference in the time, either this can happen, the message can go very fast, or sometime it will be really slow. There's a big variation in here. This happen basically because of the network communications.

How many of you know a little bit about TCP? OK. Let me just explain about five minutes of TCP before I move on. So TCP is one of the main protocols that we use to communicate over Internet. And two things, you want to actually send data. But also, you want to be a good citizen. You have actually work in a way that it doesn't really take over the entire shared bandwidth you have. So what TCP does, it has a window. So what it says is, OK, I can send certain amount of data that's the size of the window, but I can't move beyond that until I get some acknowledgement. So I send the window amount of data. And, on the other side, when it's received that data, I said, I have seen this much of the window. And once it's seen this much of the window this send that acknowledgement back. And when that acknowledgement comes, you say, aha, that's good. That means it has seen that. Then I can send more. I keep sending more.

And then, the TCP has this very interesting property. If you are doing a really good communication, things are going very nicely, it starts increasing the window size. It says, oh, OK, that means that windows size keeps going. I can do bigger and bigger and bigger window size. You can keep increasing the window size. And then, what happens at some point, the system get overloaded, because if you everyone is start-- increase their window size, too many packets start coming into the network. At some point, the network in the middle, doesn't have enough room. It drops something. So, TCP, nice thing about the network is that, even if it doesn't have a guarantee that it will guarantee it can just drop something. And when it drops, what happens is the other guy is waiting for acknowledgement. So waiting for data to come, it never shows up. So then it has this thing, an ack, saying I never got it. And the problem with that is, so you have a nice bandwidth, and you get increasing the bandwidth, you get faster and faster and faster, and suddenly data get missed. And suddenly you have this big timeout delay. And then everybody freezes. And then get the ack, and you restart with a smaller window and slowly pick up, something like that. So because of that, there a lot of times what happens, is this packet get dropped, retransmit happens, so you have this kind of sawtooth pattern. Things get faster and faster, things go down for nothing, for a little while, again start again, in here.

So the other way of communicating is called UDP. UDP says, OK, if you don't have any kind of acknowledgement, or something like that, I'll just send. And you, on the other hand, figure this out whether you got something or not. And send information back. So the network doesn't participate in any kind of a balancing act of communication. It's end-to-end. So of course, you can be a really bad citizen, and say, OK, I don't care. I just keep sending huge amount of data and then somebody would go well. But, what people have found is for things like video, you can send UPD and kind of manipulate yourself end-to-end, can get much better than trying to do this TCP. So the kind of thing is, even though there's not acknowledgment, or there's no real attempt to make sure all the data goes, UDP sometimes can get better bandwidth, because it doesn't drop packets in here. So, there's a lot of great stuff, I mean you guys can take the networking class and learn all about the protocols and stuff like that. There's really, really cool stuff in here, so some of you might actually, next couple of semesters, learn all about how these things work. So I'm just giving you, lot of performance wise, these are the issues that you are to worry about when you are doing network level things.

OK. So that's kind of talks about a little bit about a small scale, and then if you want to go to next bigger scale-- why you want to go? There can be lot more uses. If you are on something like Facebook, or Amazon, you have a lot more users to deal with. If you are, what's a good one with a lot of data? Google Earth, or something like that. You have a lot of data. And you have to deal with all the data, and that's a good way to do the scale up. Or you might have huge amount of processing you want to do, for example, the one place a lot of data and processing is things like these new basically telescopes that's coming about, that has arrays of hundreds of different things, so you have huge amount of data coming from the telescopes. And then you could do a huge amount of processing and that. So that basically, has broad data and processing, and in things like webs, social networks, and stuff like that, gives me a lot of data.

So here are some examples of some things like the airline reservation system. It's something, all the airlines have to assign millions of planes, flights, millions of seats that you deal with. Things like a stock trading system that all the trades has to has to come there, and the prices has to get calculated, and then trades has to get validated. And, very big analysis, so you form some kind of global understanding of what's going on. And I'm going to talk about these two, three things too. Scene completion and web search, which probably everybody knows. So, yes, this kind of data, now, kind of a web analysis example.

So what these guys were trying to do was, every weekly, troll 151 million web pages, and get about a terabyte of information, and analyze page statistics. So that's what they are trying to do. Some come up with some idea about OK, what is the world happening? How did the pages change the last week? And then try to get a global view of that. At this point, you have both huge amount of data and pretty large amount of computation power, that you had to build a system to do that. This is where you need a larger system.

Here's another interesting system that people built. So if you have image here, and the image has this nice background, there's unfortunate house sitting in the front. So what this says is, OK, eliminate the house, search a very large database to find similar images and plop something in there. OK. So OK. You can get your face with some nice actual eyes, or something like that. Just eliminate all the bad parts, and then and then get good parts and put them in there. And so this one, basically, what they'll do was, that's about 396 gigabytes of images out there. And so we had to classify images to get the scene detector, do color similarity, and do context matching. So computation, what they're doing is about 50 minutes doing scene matching, 20 minutes of local matching trying to find right matching, and four minutes composing there, and then you can parallelize that and reduce this time to about five minutes. So here's something that's huge amount of data. You'll look a lot of things, you do a lot of processing to figure out we get the right thing and these actually keep increasing these images as we keep asking for more, more flexibility, and more accuracy. Things can get higher and higher.

So really cool application that really require large data and large processing. So, of course, the kind of clinical application is probably Google. So in this research, you'll get some nice results. So what people say, is this what two thousand process involved getting this query for you. It takes 200 plus terabytes of data, but this is already old now, this could be even higher now. And this takes ten to the ten total clock cycles for everything that needs to happen, for you to get to your query. And you only get one sent for the query. So that, not only are doing it fast, you are doing a lot of processing, you are doing it very cheap. And I think one of the biggest things that Google did is figure how to get that done fast and cheap. And that's why they so successful. Oops, sorry, I didn't say this. so it's one second response time, and the cheapest $0.05 average the cost, basically. If you compute a time that's going to cost more than $0.05, is not worth it. And you had to do it that. So, this is Google, spend a lot of time how to figure out, how to do this is cheaply.

So, this is already validated, but Google is very secretive of what they do, so this is the closest I can figure out. They have three million plus processers in clusters of 2000 plus process, each, in each cluster. And what they already did was they went for the cheapest thing. They build entire system out of the cheapest parts we can get. x86 processors, the cheapest disks, fairly cheap communication, and gain reliability, redundancy though software. So each part, I mean supposing in Google, this data center, there's somebody who's constantly growing and changing machines and changing disks. Because there's so much failure. But that means we have to have the software system to keep the things running in there. And what they have is a partitioned workload, all those things are nicely partitioned and distributed through Google as this nice file system and stuff do that. And then you have to do crawling, index generation, index search, document retrieval, ad placement, all those things happen in there. Of course, other things like Microsoft and Yahoo, and all those other people have systems like that. So this is kind of what, when you go in here to scale up, there's no other way, you have to actually build this huge system to do that.

So one thing Google does, going a little bit technical, is this have a system called MapReduce. How many of you have seen, heard of MapReduce? OK. So there's all this people who know MapReduce. Probably more than I do. So the idea there is you have a bunch of data, a huge amount of data in here. And, normally, what you have to do is find some similarities in lot of data, and do some processing for that. And this is programming model set up nicely help doing that. So, that this borrows lot of functional programming. What that means is I'm not changing data, I'm always taking some data values and creating something new. I'm never changing something existing, that's basically meaning of a functional program.

So MapReduce has two components. First the map. That means given some input value and a key in there, what you develop generate is some intermediate results and output key. You get bunch of values coming through, and everybody process each one as separate. And say, OK. So here is the output key, and here's some intermediate value. And then what you do is things with the same output key gets sorted into one list. And then it's going reduce it. And the reducer takes the output key in this list, and say, OK, look I'm going to process the entire list down to one element or small data item. OK?

So let's go through a little bit more, digging deep into that. And so you map, basically get a huge amount of records from the data source, and it fits into this map function, and it produce intermediate results. And the reduced function, basically, combines the data, and all the folding-- let me give you an example. I think that will show you better. So here is kind of architecture. So you have a huge amount of data resources. You have many, many sources in here. And each of the data comes in to that, and the map will basically distributed by keys and values, so there could be millions and values. And then, what you have to do is, wait until all the data, has done that. And then cleared for the number of keys here, number of reducers. So hopefully you wont have a lot of keys. If you have more than two keys, you don't get that parallelism because then you would be too huge lists. And then, again, what happens is these keys get paired to reducers to come the final value in here.

So what's the parallelism here? What makes the parallelism go high? Or, not have enough parallelism? Yeah, I mean, first of all, you need to have enough, hopefully, multiple data stores so you get a lot of parallelism coming in here. Map is easily parallelizable, because each choosing in here. Reducer is the problem I think one. Because if you have too many keys, too little keys, you are in trouble. The other interesting thing in here is there's a big shuffling between here. So that means data has to go all over the place. So it's not something that, you got data and you process to the end you got data you process to end, every data has to kind of cross back, and so that's a huge amount of communication in here that could be bottle-necked too. That can be bottle-necked, keys can be bottle-necked. So map function runs parallel, creating different things. Reduced functions also run parallel for each key. And all values are basically processed independently because of that. Also, the bottle-neck is reduce phase can't start until all the map is done, and also all the data gets shuffled around with that.

So here's an interesting example. What I am trying to do is I am trying to count the number of words in assume huge amount of web pages. So what I can do is in the map, I get each page in here-- I thread through the page emitting each word as my key and the count as one. Because I only get one thing.

And then my reducer is basically-- my key is each word. So if I have a million words, I can have a million reducers. And the reducer basically takes all those things-- it's not that fun because everything is all at number one. Because we count at one.

And then basically keep adding up how many things for each word came about and put up the results. So you can say, OK, look, for the entire corpus of data, I had this many words count, this many word occurrences, this many all for each word. You get a word count.

So basically trying to create a histogram here and MapReducer provides a very nice interface to do that. And it's very nice, high level, and it provides this nice infrastructure to run this in parallel in here and do all the communication necessary, figure out how many reducers to run, look at machines to run them, produce the result, and give you the result. So this is a nice infrastructure Google has built in there.

So in this level, when you go to this-- this is the data center level. What do you have to do to scale? You need to distribute data. And you need to parallelize because if all the data is in one machine, it doesn't help.

And you need to have parallelism to scale everything. Another interesting thing you can do is approximate. So what that means is normally when you calculate, when everybody has exactly the same data all the time-- because when you write the memory, everybody sees that memory-- you have the perfect knowledge of the word.

And in a distributed system, getting perfect knowledge is very expensive. That means every time something changes, you have to send everybody that data. And one way that people really make these systems run fast, you say, wait a minute, if somebody doesn't have the perfect knowledge, if there's a little bit of discrepancy between something, I am OK.

Assume you have a new-- you changed your-- we'll say-- web page and added a couple of new words in there. Next second, the search doesn't see it. Nobody's going to complain. And then you can deliver that.

If you do a search, you'll find something. But somebody else doesn't do it because that data haven't propagated that to both of the things. Nobody's going to complain and say, wait a minute, I found it, but he didn't.

It can have a little bit of a lag. And that can be really, really useful in these kind of systems. Because every time something happened, you don't have to keep updating. But tell me a system that you can't actually do that. Play it a little bit fast and easy.

AUDIENCE: Stock trading.

PROFESSOR: Stock trading. Yeah, that's something basically, if you say, yeah, you might get it too, you might get it-- and that doesn't work. Basically, stock trading has this very particular thing because when it does submit, we'll say, a sale order, within a certain amount of time, it has to get matched up and has to be announced to both people.

And also, there are a lot of other constraints like if the machine goes down. Either trade has to be everybody saw the trade or nobody saw it. You can't say-- somebody says, I sold it. And another guy says, no, I didn't buy it.

And when you have millions of billions of dollars back and forth, that doesn't really work. So for that, there's this thing called transactions. So transactions is an interesting way-- a lot of databases have this transaction.

Transactions say, look, I am doing this very complicated thing. And I cannot have this intermediate state going on. So what transactions say is, first, tell me everything I want to do in the transaction.

So it might be I want to sell a stock, I want to buy a stock, whatever. And then at some point when you commit the transaction, you either say, OK, everything worked, good, the entire thing gets committed. And you are done.

Or it can explicitly reject it. It can come back and say, look, I can't do this transaction. Now sorry, you can restart it. So you can accept and reject.

But then the nice thing about that is then every one single-- it's like atomicity. Every one action doesn't have to happen immediately or happen as a group. You can say, OK, I'm doing a bunch of action in the transaction. And then finally, I can come in and if it works, great.

So what might be a reason you might not be able to commit a transaction if you do a transaction? Anybody else want to answer? When you say, I want a transaction, I want to commit something, what might say-- so here's an interesting thing. In the stock trading type world-- so assume I want to sell something.

Let's look at the airline reservation. So assume I have an airline seat in here. And if two people want to try to resell that seat, I can do all this processing in parallel for everybody until I come to the commit point. That means I can look at everybody after you enter the data, do the price, all those things separately for the same seat.

But then when you come to the commit point, you say, can I commit the transaction? At that point, only at that point, they have to figure out whether there's a conflict in here. And at some point, if there's a conflict, it says, oops, can't.

One transaction has to get aborted. The nice thing about that is most of the time people are not going to fight for the same seat. And then things can proceed in parallel. You don't have to wait.

Otherwise, if you do that, there might only one seat assignment at a time you can do. And that's really not going to scale. So everybody tried to get their seat. They go to the end, and they say, can I proceed?

And at that point, you check whether there's a conflict. And if there's a conflict, one guy backs out. So that's the transaction. Oops, I'm going to get rebooted I guess.

So when you go to planet scale, you can get even into more issues, things like-- what could be a planet scale thing out there? What's an interesting planet scale thing that you can think of? Single computation that has to happen in the planet scale.

Something like Internet naming system. It has to work everywhere in the entire planet. Or something like Internet routing. There has to be an algorithm that has to work.

The entire world has to cooperate and then make sure that all the traffic actually goes to the right place. So there's a lot more issues, interesting things show up in here. So things like Seti@Home type stuff-- these are a little bit dated these days, that happens-- distributed all across the place.

So if you do planet scale, it has to be truly distributed. There cannot be any global operations, no single bottleneck. And you have to have distributed view with stale data. You cannot say, look, everybody has to have the same data. You have to have everything distributed.

And it has to add up to load distributions because things can keep changing in there. So what I'm going to do next is trying to give you a little bit of a case study that shows you some interesting properties that show up when you start building at that scale. And this has some planet scale type properties, some cluster properties, whatever. And I will probably first describe this interesting problem and then show what kind of solutions that came through, so to give you a perspective for a problem in here.

Any questions up to this far for distributed systems? It's hard to do distributed systems in one lecture. There are almost closest for distributed systems. But this will give you a feel for some of it.

So the case study here is from VMware. It's called deduplication at global space. And the problem shows up when you're trying to move virtual machines across the world. You have this virtual machine.

So what virtualization did was it took a piece of hardware and it converted it into a file. So each machine is now a file. When you have a file, like hardware, there are a lot of cool things you can do then.

You can replicate those files. So suddenly, instead of one machine, you've got tens of hundreds of machines. You can move those things in here.

And of course, you can start another machines all over. So once you are able to move these things, the issue becomes how to move those things around and what's the cost of moving something. And also, you can store it, store those things in a database.

The interesting thing that's happening these days is cloud computing. Cloud means there's all these providers all over the place, saying, I have processing power, I can give you some of them. Amazon does something easy too, but Verizon is trying to do, everybody is trying to do that. So if you want to have the best market, what you want to do is have the elasticity to move from cloud to cloud for many reasons.

So sometimes the cloud might be too small. You want to get to a bigger cloud in there. Or you want to be near the users.

So in the daytime in the US, you want probably to move the machines through US. At night, there might be users in China, so you want to move your compute nearer China. Because it will be closer to the people who are using it. So something like that, you can move around there. Or you want to find the cheaper provider. If somebody comes and says, look, I can give you compute power $0.10 cheaper than what you are getting, OK, I want to move to that guy.

And also, to amortize the risk of catastrophic failure. If there's a hurricane approaching somewhere, I might want to move to a data center that might be out of the way. And I want to do that. And the interesting thing there is a lot of things.

But when you say, application in the cloud, it's a machine, a machine basically in a virtual machine. At the same time, a virtual machine has to get moved around, not your small application. The entire thing has to move around.

And virtual machines are hefty. Because it has an operating system, it has all the software. There's so many things now. Then your data and your state.

There are a lot of things in the machine in here. And all those things have to get moved around, so that can be expensive. So yes, interesting experiment in here.

So the idea here is to try to move something from Boston to Palo Alto on a 2 megabytes network. And there are a bunch of different virtual machines in here, VMs. And it takes-- whatever-- 3,000 minutes to move these machines from-- this is 500 minutes. That means what? Some hours to move them. Some hours to move these machines around.

And then what you say, look, the machines are heavy, big. Why can't you first compress the machine? So you can use something like normal compression.

So blue is basically compress, move the machine, and decompress. So here is something interesting. This is a very fast compression.

You did really well, you are moving there. If you want a better compression, you say, I'm going to do a full, best compression I can do, it's actually slower. Because the trouble is the compression time is so high, the reduction is not usable in here.

So you try to compress, you spend most of the time compressing. So actually, this is even slower that just sending without compression. So compression is important, compression is useful.

So can you do better than a normal compression in here? How can you do better? So some key observations in here.

So a large part of these files are executables. You have your Linux kernel, whatever, to all those executables hitting in there. And basically, that's monoculturing the world.

There are no million different executables. You are a Linux kernel, there's only a certain amount of versions. Microsoft XP, there are certain types of versions.

So even though there are millions of machines, inside the millions of machines, there aren't millions of different applications. There's only hundreds of different applications. And so your motion moves.

If you think about it, you are moving the same thing again and again and again. Can you take advantage of that? And there's even substantial redundancy in each of these.

So this is very interesting. If you have a Windows machine, each DLL has three copies. So you have the copy, and then the Installer has a copy. And then there's another copy in the next version to basically back out, so undo copy.

So each thing is kept three copies. So every big thing, they're seeing multiple copies in there. So that part is there also. Even within a single disk, there is redundancy in here.

And another interesting thing is many of the disks have a large amount of zero pages. So if you send something uncompressed, you send a huge amount of zeros. So you are waiting for zeros to get in there. Even easy compression can get to those zeros, but this is a large chunk of data in here.

And so the interesting thing is if you take one virtual machine, this is the number of non-zero blocks. And this is the number of unique blocks. So unique blocks are smaller than non-zero blocks.

But if you keep adding more and more virtual machines, then of course, the number of total blocks keeps going up. But the unique blocks doesn't keep increasing. That means the second Linux box you add, there's not much new in there.

So if you look at that, what happens is the first guy has about 80% things are unique. When you keep adding things, it's about only 30% is unique after you add. Because it's the same program. Only the data is different as you keep adding.

So can you really take advantage of that? So that is where deduplication comes in. So deduplication says, I have this data, I have a lot of redundant data.

So A B, A B, A B is redundant. So what you want to do is break it up to some kind of blocks in here. And then one easy way to do is calculate a hash.

Because you don't want to compare blocks. That's too much. N squared comparison of blocks is a lot of comparison. You can have some kind of hash calculated for each of these blocks.

And then you can compare the hashes. And if the hashes are the same, they are the same blocks. And then what you can do is you can eliminate most of these blocks in there and then keep hashes for each block-- only the hash.

And then what you can do is you can only keep the unique blocks in here. So even though you have nine in here, only five different unique blocks are there. So that's a nice way of deduplicating. So you actually have what you call recipe, a common block store in here.

So one way to do that is you can have a recipe on common block store for each of the systems in here. That's the tradition of deduplication. Or what you can do is have everybody keep a recipe and only have one common block store.

Just keep one, single common block store, and everybody have a recipe, or probably cache of a recipe. So by doing that, you can even reduce a huge amount of the things happening in here.

So the interesting thing is if you are keeping one common block store, who can keep, who can manage? That's the interesting question in here. So can you keep instead of common block store for each processor, each computer, can you keep the common block store for the entire world?

So if you find most of the common blocks in the world, keep one store. And the nice thing about that is then I can go anywhere in the world. I can ask for the common blocks for the common things. I can populate it myself.

So here's this interesting system called the Bonsai. What they did was-- so if you have a block in here, you calculate a hash function in here, get a hash key. So what that means, the hash key can uniquely access this block. And then what you want to do is ask that you want to compress this block. This additional step, I'll explain later why it's needed.

So the other thing you can do is you can get a second hash key and use that as a private key to encrypt this block. Because you calculate two hash keys. One is the hash key to identify. The other one is a private key to encrypt this block.

And then what you can look at is this global store to see whether this hash key exists. If the hash key exists, then say, I got the page, here is the page. That's the encrypted page. And each page will have a unique ID.

And so here's my unique ID. If you find the page in here, what you can do is you can only store UID and this private key and get rid of my page. So storing UID and private key is sufficient to get my page and unencrypt it.

Why do you think I have to compress here? Why do I have to basically do encrypt here? What's the interesting thing about encryption? Why encrypt?

Assume this is global. Because if you don't encrypt, it might be a common page, but it might not be something you want everybody to know. So assume a large company like Google. President of Google, Larry Page, sends everybody emails, saying, this is very private, but here is something that's happening in the company.

And it will get into everybody's mailbox. And suddenly, it becomes-- aha-- a common page. And it gets sucked in the world because of the common page. And now everybody can see that, and that's not good.

But now if you have this private key-- if you don't have the private key, I can't decrypt that. So what happens is-- let me go through what you can do in here. And then what can happen is if you had these two, UID and private key, you can go to the global storage and say, here's the UID, give me the page.

It has a page. It better have the page for that UID. Get the page out of that. And then now I can use my private key to decrypt it. And then of course, I can decompress it in the original page.

So by doing that, I can have this global system that keeps common pages in there and store them. I can basically, really eliminate the storage. Because the nice thing is this could be, we'll say, 2K pages. And this is 64 bit, and this is probably 256 bit. So there's a huge compression of data in there.

So here are the kinds of decisions you are to make. For example, hash key. So each page is represented by a hash key. But you can have two hash keys, two pages mapping to the same hash key. So my god, you're not unique. So why is it OK?

AUDIENCE: Low probability.

PROFESSOR: Very low probability. And in fact, what they looked at was they calculated the disk failure. There's a higher failure of disk failing and losing data than hash collision.

So you can say, look, you're keeping that in the hard drive, so there could be more chance of disk failure than hash collision. So therefore, hash key is-- you can do that, low enough probability, you can get away with that. Actually, I want to skip the rest of this in there.

So here is the comparison how much compression you can give. Some of them, you almost can compress 100%. Because if you have a newly installed Linux box with very little data, everything is in the global.

So by doing that, basically, here is the communication. The compression is cheap. You don't have to do too much because you just do this hash comparison to do that.

And then now the communication time is even reduced because you communicate a lot less. And then you expand. So all these three are actually now much faster to send a machine across.

So here is a total size of all the VMs. The interesting thing is if you compress within, if you just do compression, most of this is zero blocks. You can eliminate the zeros and do a simple compression in here.

And if you do local deduplication, if you eliminate that, and if you do global, you'll get to this point. So you get to about basically 30% of all your data. And yeah, there's more data in here, but just--

So that's an interesting global level system that people are building today. And I think if these kind of systems appear in many different places. These days, a lot of people are building things like cell phone games and things like that, has this huge back stores, back computation. All those things have this large scalability in there. And so the nice thing about performance engineering is if your application doesn't require a huge amount of computation or very fast processing, if you're going to have millions and millions of users, or if you're expecting millions of users, then building these systems, and understanding, and building scalability is really important. And a lot of the things we learned in this class is directly applicable there.

So I have...

Any other questions before you guys can go finish your project report and go have a nice Thanksgiving dinner? So everybody is thinking that they will be able to get a good handle on what they are doing for the final project? Oh come on, there are so many cool things you can do with this project.

AUDIENCE: We're working on it.

PROFESSOR: And there are-- whatever-- three or four iPod Nanos waiting. So it makes a lot of sense to actually really focus this one. This is a fun project. This is, in fact, a fun project.

Because talk to these guys what they did last year. People did a lot of interesting things. Because we gave you freedom to actually even change the algorithms. So you can actually look at the algorithm.

If you know a little bit of physics and graphics type stuff, look at them and say, look, can I even reduce computation of how the computation is being done? So people got a lot of wins by doing things like that. And of course, parallelization matters. And there are a lot of optimization possibilities in this piece of code.

So take a look at that. Just have a plan. Just don't go blindly into it, just have a plan. Run it, profile it, get some feedback. You need to get this for your presentations.

So run, profile, get some feedback, have a good plan. Go attack it. So see you in a week.